DRAFT NIST Special Publication 800-178

A Comparison of Attribute Based Access Control (ABAC) Standards for Data Services

Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC)

David Ferraiolo
Ramaswamy Chandramouli
Vincent Hu
Rick Kuhn

COMPUTER SECURITY

National Institute of Standards and Technology
U.S. Department of Commerce

DRAFT NIST Special Publication 800-178

A Comparison of Attribute Based Access Control (ABAC) Standards for Data Services

Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC)

David Ferraiolo
Ramaswamy Chandramouli
Vincent Hu
Rick Kuhn
Computer Security Division
Information Technology Laboratory

December 2015

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

Authority

National Institute of Standards and Technology Special Publication 800-178
Natl. Inst. Stand. Technol. Spec. Publ. 800-178, 57 pages (December 2015)
CODEN: NSPUE2

Public comment period: *December 2, 2015* through *January 15, 2016*

All comments are subject to release under the Freedom of Information Act (FOIA).

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-178@nist.gov

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) are very different attribute based access control (ABAC) standards with similar goals and objectives. The aim of both is to provide a standardized way for expressing and enforcing vastly diverse access control policies on various types of data services. However, the two standards differ with respect to the manner in which access control policies are specified and implemented. This document describes XACML and NGAC, and then compares them with respect to five criteria. The goal of this publication is to help ABAC users and vendors make informed decisions when addressing future data service policy enforcement requirements.

Keywords

access control; access control mechanism; access control model; access control policy; attribute based access control (ABAC); authorization; Extensible Access Control Markup Language (XACML); Next Generation Access Control (NGAC); privilege

Acknowledgements

The authors wish to thank their colleagues who reviewed drafts of this document. The authors also gratefully acknowledge and appreciate the comments and contributions made by government agencies, private organizations, and individuals in providing direction and assistance in the development of this document.

Trademark Information

Executive Summary

Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) are very different attribute based access control (ABAC) standards with similar goals and objectives. XACML, available since 2003, is an Extensible Markup Language (XML) based language standard designed to express security policies, as well as the access requests and responses needed for querying the policy system and reaching an authorization decision [17]. NGAC is a relations and architecture-based standard designed to express, manage, and enforce a wide variety of access control policies through configuration of its relations. Commonly asked questions are, what are the similarities and differences between these two standards? What are their comparative advantages and disadvantages?

These questions are particularly relevant because XACML and NGAC are different approaches to achieving a common access control goal—to allow data services with vastly different access policies to be expressed and enforced using the features of the same underlying mechanism in diverse ways. These are also important questions, given the prevalence of data services in computing. Data services include computational capabilities that allow the consumption, alteration, and management of data resources, and distribution of access rights to data resources. Data services can take on many forms, to include applications such as time and attendance reporting, payroll processing, and health benefits management, but also including system level utilities such as file management.

To answer these questions, this document first describes XACML and NGAC, then compares them with respect to five criteria. The first criterion is the relative degree to which the access control logic of a data service can be separated from a proprietary operational environment. The other four criteria are derived from ABAC issues or considerations identified by NIST Special Publication (SP) 800-162 [13]: operational efficiency, attribute and policy management, scope and type of policy support, and support for administrative review and resource discovery.

Although NGAC is only now emerging as a national standard, it compares favorably in many respects with XACML and should be considered, along with XACML, by both users and vendors in addressing future data service policy enforcement requirements. Below is a summary of this comparison.

Separation of Access Control Functionality from Proprietary Operating Environments

Both XACML and NGAC achieve separation of access control functionality of data services from proprietary operating environments, but to different degrees. XACML's separation is partial. An XACML deployment consists of one or more data services, each with an operating environment-dependent policy enforcement component, and operating environment-dependent operation and resource types, that share a common policy decision function and access control database consisting of policies and attributes. The degree of separation that can be achieved by NGAC is near complete. Although NGAC issues application and system utility-specific access requests, these requests may be comprised of operations that consist of sequences of standardized operations on data resources and NGAC's access control data. The requests are issued through a standardized enforcement component to a standardized decision component, with functionality that is not dependent on an application operating environment.

Operational Efficiency

An XACML request is a collection of attribute name, value pairs for the subject (user), action (operation), resource, and environment. XACML identifies relevant policies and rules for computing decisions through a search for Targets (conditions that match the attributes of the request). Because multiple Policies in a PolicySet and/or multiple Rules in a Policy may produce conflicting access control decisions, XACML resolves these differences by applying collections of potentially twelve rule and policy combining algorithms. The entire process involves collecting attributes, matching conditions, computing rules, and resolving conflicts, involving at least two data stores.

NGAC is inherently more efficient. An NGAC request is composed of a process id, user id, operation, and a sequence of one or more operands mandated by the operation that affects either a resource or access control data. NGAC identifies relevant Policies and attributes by reference when computing a decision. NGAC computes decisions by applying a single combining algorithm over applicable Policies that do not conflict. All information necessary in computing an access decision resides in a single database.

Attribute and Policy Management

Proper enforcement of data resource policies is dependent on administrative policies. This is especially true in a federated or collaborative environment, where governance policies require different organizational entities to have different responsibilities for administering different aspects of policies and their dependent attributes.

XACML and NGAC differ dramatically in their ability to impose policy over the creation and modification of access control data (attributes and policies). NGAC manages attributes and policies through a standard set of administrative operations, applying the same enforcement interface and decision making function as it uses for accessing data resources. XACML does not recognize administrative operations, but instead manages policy content through a Policy Administration Point (PAP) with an interface that is different from that for accessing data resources. XACML provides support for decentralized administration of some of its access policies. However the approach is only a partial solution in that it is dependent on trusted and untrusted policies, where trusted policies are assumed valid, and their origin is established outside the delegation model. Furthermore, the XACML delegation model does not provide a means for imposing policy over modification of access policies, and offers no direct administrative method for imposing policy over the management of its attributes.

NGAC enables a systematic and policy-preserving approach to the creation of administrative roles and delegation of administrative capabilities, beginning with a single administrator and an empty set of access control data, and ending with users with data service, policy, and attribute management capabilities. NGAC provides users with administrative capabilities down to the granularity of a single configuration element, and can deny users administrative capabilities down to the same granularity.

Scope and Type of Policy Support

Although data resources may be protected under a wide variety of different access policies, these policies can be generally categorized as either discretionary or mandatory controls. Discretionary access control (DAC) is an administrative policy that permits system users to allow or disallow other users' access to objects that are placed under their control. Although XACML can theoretically provide users with administrative capabilities necessary to control and give away access rights to other users, the approach is complicated by the need to create and maintain additional metadata for each and every object/resource. Conversely, NGAC has a flexible means of providing users with administrative capabilities to include those necessary for the establishment of DAC policies.

In contrast to DAC, mandatory access control (MAC) enables ordinary users' capabilities to execute resource operations on data, but not administrative capabilities that may influence those capabilities. MAC policies unavoidably impose rules on users in performing operations on resource data. MAC policies can be further characterized as controls that accommodate confinement properties to prevent indirect leakage of data to unauthorized users, and those that do not.

Expression of non-confinement MAC policies is perhaps XACML's strongest suit. XACML can specify rules and other conditions in terms of attribute values of varying types. There are undoubtedly certain policies that are expressible in terms of these rules that cannot be easily accommodated by NGAC. This is especially true when treating attribute values as integers. For example, to approve a purchase request may involve adding a person's credit limit to their account balance. Furthermore, XACML takes environmental attributes into consideration in expressing policy, and NGAC does not. However, there are some non-confinement MAC properties, such as least privilege, and a variety of history-based policies that NGAC can express, which XACML cannot.

In contrast to NGAC, XACML does not recognize the capabilities of a process independent of the capabilities of its user. Without such features, XACML is ill equipped to support confinement and as such is arguably incapable of enforcement of a wide variety of policies. These confinement-dependent policies include some instances of role-based access control (RBAC), e.g., "only doctors can read the contents of medical records", originator control (ORCON) and Privacy, e.g., "I know who can currently read my data or personal information", or conflict of interest, e.g., "a user with knowledge of information within one dataset cannot read information in another dataset". Through imposing process level controls in conjunction with event-response relations, NGAC has shown [7] support for these and other confinement-dependent MAC controls.

Administrative Review and Resource Discovery

A desired feature of access controls is review of capabilities of users and access control entries of objects [11]. These features are often referred to as "before the fact audit" and resource discovery. "Before the fact audit" is one of RBAC's most prominent features [18]. Being able to discover or see a newly accessible resource is an important feature of any access control system. NGAC supports efficient algorithms for both per-user and per-object review. Per-object review of access control entries is not as efficient as a pure access control list (ACL) mechanism, and per-user review of capabilities is not as efficient as that of RBAC. However, this is due to NGAC's consideration of conducting review in a multi-policy environment. NGAC can efficiently support both per-object and per-user reviews of combined policies, where RBAC and ACL mechanisms can do only one type of review efficiently, and rule-based mechanisms such as XACML, although able to combine policies, cannot do either efficiently.

Table of Contents

List of Appendices

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope

The purpose of this document is to compare and contrast Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) — two very different access control standards with similar goals and objectives. The document explains the basics of both standards and provides a comparative analysis based on attribute based access control (ABAC) considerations identified in NIST Special Publication (SP) 800-162, *Guide to Attribute Based Access Control (ABAC) Definition and Considerations* [13].

1.2 Audience

The intended audience for this document includes the following categories of individuals:

- Computer security researchers interested in access control and authorization frameworks
- Security professionals, including security officers, security administrators, auditors, and others with responsibility for information technology (IT) security
- Executives and technology officers involved in decisions about IT security products
- IT program managers concerned with security measures for computing environments

This document, while technical in nature, provides background information and examples to help readers understand the topics that are covered. The material presumes that readers have a basic understanding of security and possess fundamental access control expertise.

1.3 Document Structure

The remainder of this document is organized into the following sections:

- Section 2 provides background information on the origins, makeup, and objectives of XACML and NGAC.
- Section 3 describes XACML's policy specification language and reference architecture for ABAC implementation.
- Section 4 describes NGAC's fundamentally different approach from XACML for representing requests, expressing and administering policies, representing and administering attributes, and computing and enforcing decisions.
- Section 5 provides an analysis of XACML and NGAC's similarities and differences based on five criteria.
- Appendix A provides a list of acronyms used in the document.
- Appendix B contains a list of references.
- Appendix C provides a formal XACML policy specification for an abbreviated policy example in Section 3.

2 Background

XACML and NGAC both provide attribute-based approaches to accommodate a wide breadth of access control policies and simplify their management. Most other access control approaches are based on the identity of a user requesting execution of a capability to perform an operation on a data resource (e.g., read a file), either directly via the user's identity, or indirectly through predefined attribute types such as roles or groups assigned to that user. Practitioners have noted that these forms of access control are often cumbersome to set up and manage, given their limitation of associating capabilities only to users or their attributes. Furthermore, the identity, group, and role qualifiers of a requesting user are often insufficient for expressing real-world access control policies. An alternative is to grant or deny user requests based on arbitrary attributes of users and arbitrary attributes of data resources, and optionally environmental attributes that may be globally recognized and tailored to the policies at hand. This approach to access control is commonly referred to as attribute-based access control (ABAC) and is an inherent feature of both XACML and NGAC.

From a policy management perspective, ABAC has advantages over other access control approaches. ABAC avoids the need for capabilities (operation, data resource pairs) to be directly assigned to every instance of a user or resource before the request is made. Instead, when a user requests access, the ABAC engine (depicted in the center of Figure 1) can make access control decisions based on the assigned attributes of the requesting user and data resource instances, environmental attributes, and a set of policies that are specified in terms of those attributes. Under this approach, policies are managed without direct reference to potentially numerous users and data resources, and users and data resources can be provisioned through attribute assignment without reference to policy details.

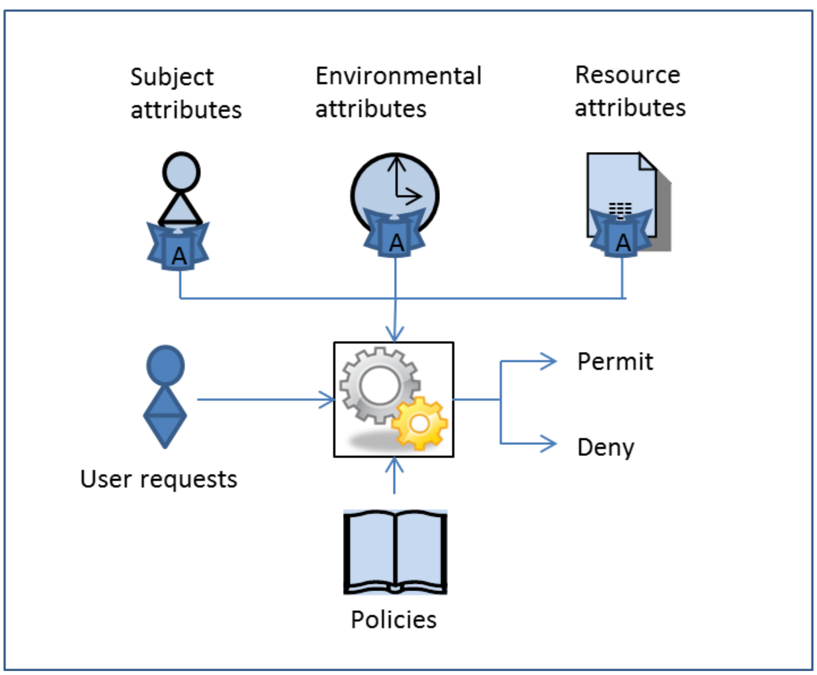

Figure 1: ABAC Overview

XACML and NGAC are ABAC standards for facilitating policy-preserving user executions of data service capabilities (data service operations on data service resources). In general, data services are both applications and system utilities that provide users with capabilities to consume, manipulate, manage, and share data. Data services can take on many forms, including applications such as time and attendance reporting, payroll processing, corporate calendar, and health benefits management, all with a strong dependency on access control. The XACML and NGAC standards, enable decoupling of access control logic from proprietary operating environments (e.g., operating system, database management system, application).

Stated another way, a data service is comprised of an application layer and an operating environment layer that can be delineated by their functionality and interfaces. The application layer provides a user interface and methods for data presentation and manipulation (e.g., font selection, spell correction), and an interface for management and distribution of access rights on data. The application layer does not carry out operations that consume data, alter the state of data, or alter the access state to data (e.g., read, write/save, create and delete files, submit, approve, schedule), but instead issue requests to the operating environment layer to perform those operations. An operating environment implements operational routines (e.g., read, write) to carry out application access requests and provides access control to ensure executions of processes involving operational routines on data resources are policy preserving. In addition, operating environments provide methods for authenticating users, creating and associating users with their processes, and managing data resources and access control data.

Access control mechanisms comprise several components that work together to bring about policy-preserving data resource access. These components include access control data for expressing access control policies and representing attributes, and a set of functions for trapping access requests, and computing and enforcing access decisions over those requests. Most operating environments implement access control in different ways, each with a different scope of control (e.g., users, resources), and each with respect to different operation types (e.g., read, send, approve, select) and data resource types (e.g., files, messages, work items, records).

This heterogeneity introduces a number of administrative and policy enforcement challenges. Administrators are forced to contend with a multitude of security domains when managing access policies and attributes. Even if properly coordinated across operating environments, global controls are hard to visualize and implement in a piecemeal fashion. Furthermore, because operating environments implement access control in different ways, it is difficult to exchange and share access control information across operating environments. XACML and NGAC seek to alleviate these challenges by creating a common and centralized way of expressing all access control data (Policies and Attributes) and computing decisions, over the access requests of applications.

In 2014 NIST published SP 800-162, *Guide to Attribute Based Access Control (ABAC) Definition and Considerations* [13] to serve two purposes. First, it provides Federal agencies with an authoritative definition of ABAC and a description of its functional components. NIST SP 800-162 addresses ABAC as a mechanism comprising four layers of functional decomposition: Enforcement, Decision, Access Control Data, and Administration. Second, in light of potentially numerous approaches to ABAC, NIST SP 800-162 highlights several considerations for selecting an ABAC system for deployment. Among others, these considerations pertain to operational efficiency, attribute and policy management, scope and type of policy support, and support for administrative review and resource discovery. This report examines and compares XACML and NGAC based on these considerations. In addition, it compares XACML and NGAC in their abilities to separate access control logic necessary to support applications from proprietary operating environments.

2.1 XACML

In 2003, with the emergence of Service Oriented Architecture (SOA), a new specification called XACML was published through the Organization for the Advancement of Structured Information Standards (OASIS). The specification presented the elements of what would later be considered by many to be ABAC. In support of controlled execution of data service capabilities, the XACML ABAC model employs three components in its authorization process:

- **XACML policy language**, for specifying access control requirements using rules, policies, and policysets, expressed in terms of subject (user), resource, action (operation), and environmental attributes and a set of algorithms for combining policies and rules.
- **XACML request/response protocol**, for querying a decision engine that evaluates subject access requests against policies and returns access decisions in response.
- **XACML reference architecture**, for deploying software modules to house policies and attributes, and computing and enforcing access control decisions based on policies and attributes.

XACML is widely recognized by both the research and vendor communities. This acceptance is evident by its implementation, in whole or part, across an increasing number of product offerings.

2.2 NGAC

In 2003, NIST initiated a project in pursuit of a standardized ABAC mechanism referred to as the Policy Machine that allows changes to a fixed set of data elements and relations in the expression and enforcement of ABAC policies. The Policy Machine has evolved from a concept to a formal specification [8] to a reference implementation and open source distribution. The Policy Machine has served as a research component in support of a family of American National Standards Institute/International Committee for Information Technology Standards (ANSI/INCITS) standardization efforts under the title of "Next Generation Access Control" (NGAC) [2], [20]. In addition to the expression and enforcement of a wide variety of access control policies [6], [7], NGAC facilities can be used to effectuate security-critical portions of the program logic of arbitrary data services and enforce mission-tailored access control policies over data services [7], [9]. Taken together, these NGAC standards define:

- A standard set of data and relations used to express access control policies and attributes, and deliver capabilities of data services to perform operations on data resources
- A standard set of administrative operations for configuring the data and relations,

- A standard set of functions, interfaces, and protocols for trapping and enforcing policy on requests to execute operations on data resources, computing access decisions to permit or deny those requests, and dynamically altering access state in response to access events.

The initial standard of the NGAC family was published in 2013. It is available from the ANSI eStandards store as INCITS 499 – Next Generation Access Control - Functional Architecture (NGAC–FA) [2]. INCITS 526 – Next Generation Access Control - Generic Operations and Abstract Data Structures (NGAC-GOADS) [20] is in the approval process, and is expected to be published in the fall of 2015.

2.3 Comparison of XACML and NGAC's Origins

While largely developed in parallel, these standards were established under different timetables and circumstances. XACML was developed as collaboration among vendors with a goal to separate policy expression and decision-making from proprietary operating environments in support of the access control policy needs of applications. XACML first appeared in 2003 and was revised in 2013 by providing support for decentralized policy management. NGAC's origin stems from the NIST Policy Machine, a research effort that began in 2003 to develop a general-purpose ABAC framework. The Policy Machine, and thus NGAC, has benefited from experimental implementation and sustained analysis, resulting in increased policy support and decreased access control dependency on proprietary operational environments.

3 XACML Specification

XACML defines a policy specification language and reference architecture for ABAC implementation. The standard encompasses requests, policies, attributes, and functions for computing decisions and enforcing policies in response to access requests to perform actions on resources.

For purposes of brevity and readability, the XACML specification is presented as a summary that is intended to highlight XACML's salient features and should not be considered complete. In some instances, actual XACML details and terms are substituted with others to accommodate a simpler and more consolidated presentation.

3.1 Attributes and Policies

An XACML access request consists of subject attributes (typically for the user who issued the request), resource attributes (the resource for which access is sought), action attributes (the operations to be performed on the resource), and environment attributes.

XACML attributes are specified as name-value pairs, where attribute values can be of different types (e.g., integer, string). An attribute name/ID denotes the property or characteristic associated with a subject, resource, action, or environment. For example, in a medical setting, the attribute name Role associated with a subject may have doctor, intern, and admissions nurse values, all of type string. Subject and resource instances are specified using a set of name-value pairs for their respective attributes. For example, the subject attributes used in a Medical Policy may include: Role = "doctor", Role = "consultant", Ward = "pediatrics", SubjectName = "smith"; an environmental attribute: Time = 12:11; and resource attributes: Resource-id = "medical-records", WardLocation = "pediatrics", Patient = "johnson". Although XACML does not require any convention for naming attributes, we sometimes use the prefixes Subject, Resource, and Env for naming the subject, resource, and environment attributes, respectively, to enhance readability.

Subject and resource attributes are stored in their respective repositories and are retrieved through the Policy Information Point (PIP) at the time of an access request and prior to the computation of the decision. XACML formally defines an action as a component of a request with attribute values that specify operations such as read, write, submit, and approve.

Environmental attributes, which depend on the availability of system sensors that can detect and report values, are somewhat different from subject and resource attributes, which are administratively created. An environment is the operational or situational context in which access requests occur. Environmental attributes are not properties of the subject or resources, but are measurable characteristics that pertain to the operational or situational context. These environmental characteristics are subject and resource independent, and may include the current time, day of the week, or threat level.

In this document we use a functional notation for reporting on attribute values with the format A(), where the parameter may be a subject, resource, action, or the environment. For example, A(*e*), where *e* is the environment, may equal 09:00 (time) and low (threat level), and A(*s*), where *s* is a subject, may equal smith (name) and doctor (role). We use a tuple notation to describe multiple attributes possessed by a subject, resource, or environment. For example, for subject *s*1 we have A(*s*1) = <smith, doctor>, where the first attribute corresponds to the name and the second one to the role possessed by subject *s*1.

As shown by Figure 2, XACML access policies are structured as PolicySets that are composed of Policies and optionally other PolicySets, and Policies that are composed of Rules. Policies and PolicySets are stored in a Policy Retrieval Point (PRP). Because not all Rules, Policies, or PolicySets are relevant to a given request, XACML includes the notion of a Target. A Target defines a simple Boolean condition that, if satisfied (evaluates to True) by the attributes, establishes the need for subsequent evaluation by a Policy Decision Point (PDP). If no Target matches the request, the decision computed by the PDP is NotApplicable.

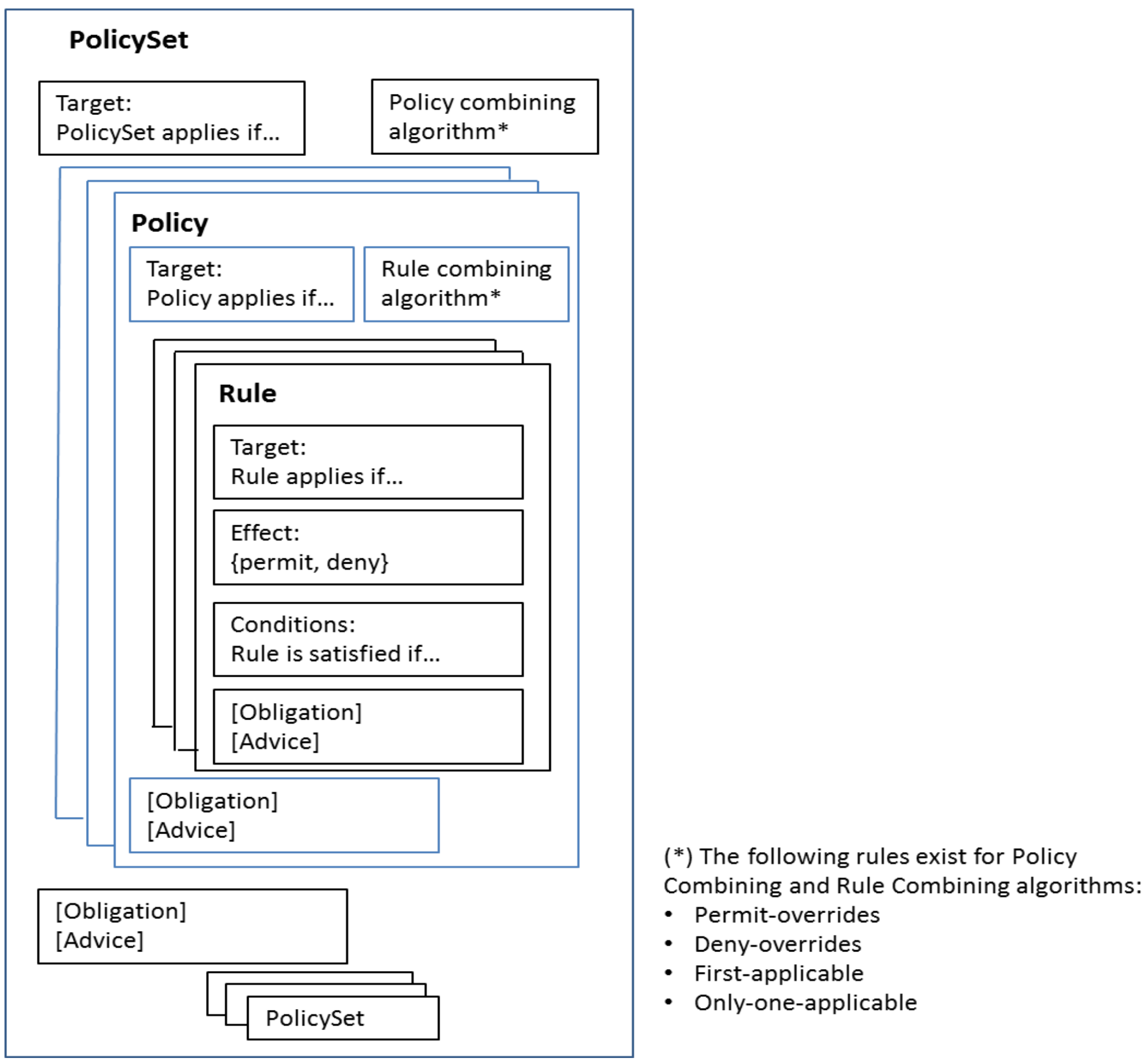

Figure 2: XACML Policy Constructs

In addition to a Target, a rule includes a series of boolean conditions that if evaluated True have an effect of either Permit or Deny. If the target condition evaluates to True for a Rule and the Rule's condition fails to evaluate for any reason, the effect of the Rule is Indeterminate. In comparison to the (matching) condition of a Target, the conditions of a Rule or Policy are typically more complex and may include functions (e.g., "greater-than-equal", "less-than", "string-equal") for the comparison of attribute values. Conditions can be used to express access control relations (e.g., a doctor can only view a medical record of a patient assigned to the doctor's ward) or computations on attribute values (e.g., sum(x, y) less-than-equal:250).

3.2 Combining Algorithms

Because a Policy may contain multiple Rules, and a PolicySet may contain multiple Policies or PolicySets, each Rule, Policy, or PolicySet may evaluate to different decisions (Permit, Deny, NotApplicable, or Indeterminate). XACML provides a way of reconciling the decisions each makes. This reconciliation is achieved through a collection of combining algorithms. Each algorithm represents a different way of combining multiple local decisions into a single global decision. There are twelve combining algorithms, which include the following:

- Deny-overrides: if any decision evaluates to Deny, or no decision evaluates to Permit, then the result is Deny. If all decisions evaluate to Permit, the result is Permit.
- Permit-overrides: if any decision evaluates to Permit, then the result is Permit, otherwise the result is Deny.
- First-applicable: the result is the result of the first decision (either Permit, Deny, or Indeterminate) when evaluated in their listed order.
- Only-one-applicable: if only one decision applies, then the result is the result of the decision, and if more than one decision applies, then the result is Indeterminate.

Combining algorithms are applied to rules in a Policy and Policies within a PolicySet in arriving at an ultimate decision of the PDP. Combining algorithms can be used to build up increasingly complex policies. For example, given that a subject request is Permitted (by the PDP) only if the aggregate (ultimate) decision is Permit, the effect of the Permit-overrides combining algorithm is an "OR" operation on Permit (any decision can evaluate to Permit), and the effect of a Deny-overrides is an "AND" operation on Permit (all decisions must evaluate to Permit).

3.3 Obligation and Advice Expressions

XACML includes the concepts of obligation and advice expressions. An obligation optionally specified in a Rule, Policy, or PolicySet is a directive from the PDP to the Policy Enforcement Point (PEP) on what must be carried out before or after an access request is approved or denied. Advice is similar to an obligation, except that advice may be ignored by the PEP.

A few examples include:

- If Alice is denied access to document X: email her manager that Alice tried to access document X.
- If a user is denied access to a file: inform the user why the access was denied.

- If a user is approved to view document X: watermark the document "DRAFT" before delivery.

A common use of an obligation, applied after an access request is approved, is for auditing and logging user access events.

It should be noted that the functionality to accommodate the directives of an obligation or advice is outside of the scope of XACML and must be implemented and executed by an application-specific PEP.

3.4 Example Policies

Consider the following two example XACML policy specifications. For purposes of maintaining the same semantics as XACML, we use the same element names, but specify policies and rules in pseudocode for purposes of enhanced readability (instead of exact XACML syntax). A more formal XACML treatment of the first policy (Policy 1) is included in Appendix C.

Policy 1 applies to "All read or write accesses to medical records by a doctor or intern" (the target of the policy) and includes three rules. As such, the policy is considered "applicable" whenever a subject with a role of "doctor" or "intern" issues a request to read or write "medical-records" resource. The rules do not refine the target, but describe the conditions under which read or write requests from doctors or interns to medical records can be allowed. Rule 1 will deny any access request (read or write) if the ward in which the doctor or intern is assigned is not the same ward where the patient is located. Rule 2 explicitly denies "write" access requests to interns under all conditions. Rule 3 permits read or write access to medical-records for "doctor", regardless of Rule 1, if an additional condition is met. This additional condition pertains to patients in critical status. Since the intent of the policy is to allow access under these critical situations, a policy combining algorithm of "permit-overrides" is used, while still denying access if only the conditions stated in Rule 1 or Rule 2 apply.

```
<Policy PolicyId = "Policy 1" rule-combining-algorithm="permit-overrides">
        // Doctor Access to Medical Records //
   <Target>
     /* :Attribute-Category    :Attribute ID     :Attribute Value */
           :access-subject      :Role              :doctor
           :access-subject      :Role              :intern
           :resource            :Resource-id      :medical-records
           :action              :Action-id         :read
           :action              :Action-id         :write
   </Target>

   <Rule RuleId = "Rule 1" Effect="Deny">
        <Condition>
           Function: string-not-equal
           /* :Attribute-Category   :Attribute ID
                 :access-subject       :WardAssignment
```

```
                :resource             :WardLocation
        </Condition>
    </Rule>

    <Rule RuleId = “Rule 2” Effect=”Deny”>
        <Condition>
          Function: string-equal
          /* :Attribute-Category   :Attribute ID     :Attribute Value
                :access-subject      :Role              :intern
                :action              :Action-id         :write
         </Condition>
    </Rule>

    <Rule RuleId = “Rule 3” Effect=”Permit”>
        <Condition>
         Function:and
          Function: string-equal
          /* :Attribute-Category    :Attribute ID            :Attribute Value */
                :access-subject      :Role                   :doctor
          Function: string-equal
          /* :Attribute-Category   : Attribute ID   : Attribute Value
                :resource            :PatientStatus   :critical
        </Condition>
    </Rule>
  </Policy>
```

Together policies (PolicySets and Policies) and attribute assignments define the authorization state. Table 1 defines the authorization state for Policy 1 by specifying attribute names and values.

Table 1. Attribute Names and Values and the Authorization State for Policy 1

Subject Attribute Names and their Domains: Role = {doctor, intern} WardAssignment = {ward1, ward2}
Resource Attribute Names and their Domains: Resource-id = {medical-records} WardLocation = {ward1, ward2} PatientStatus = {critical}
Action Attribute Names and their Domains: Action-id = {read (r), write (w)}
Attribute value assignments when there are two subjects (s3, s4) and three resources (r5, r6, r7): A(s3) = <doctor, ward2>, A(s4) = <intern, ward1>, A(r5) = <medical-records, ward2>, A(r6) = <medical-records, ward1>, and

A(r7) = <critical>.
Authorization state: (s3, r, r5), (s3, w, r5), (s3, r, r7), (s3, w, r7), (s4, r, r6)

Policy 2 applies to “IRS-agents and auditor access to tax-returns” (target of the policy) and has two rules. This policy is an “applicable policy” whenever users with role “IRS-agent or auditor” access the resource “tax-returns” with a write request. The rules do not refine the target, but state the conditions under which write requests from IRS-agents or auditors to tax-returns records can be allowed. Rule 1 will permit an applicable access request if the access time (an environmental variable) is between 8 AM and 5 PM. Rule 2 will deny the request even if the condition in Rule 1 applies through an additional condition; the IRS-agent or auditor is attempting to write to his or her own tax return. Since the intent of the policy is to disallow IRS employees from altering their own tax returns, a policy combining algorithm of “deny-overrides” is used, while still allowing access if the conditions stated in Rule 2 does not.

```
<Policy PolicyId = “Policy 2” rule-combining-algorithm=”deny-overrides”>
            // IRS Agent and Auditor Access to Tax Returns //
     <Target>
        /* :Attribute-Category   : Attribute ID   : Attribute Value */
                :access-subject      :Role                 :IRS-agent
                :access-subject      :Role                 :auditor
                :resource              :Resource-id      :tax-returns
                :action                  :Action-id          :write
      </Target>

      <Rule RuleId = “Rule 1” Effect=”Permit”>
            <Condition>
                Function: and
                /* :Attribute-Category   : Attribute ID   : Attribute Value
                        :environment       : Time               : ≥ 08:00
                        :environment       : Time               : ≤ 18:00
            </Condition>
       </Rule>
       <Rule RuleId = “Rule 2” Effect=”Deny”>
            <Condition>
                Function: and
                /* :Attribute-Category   : Attribute ID   : Attribute Value
                        :environment       :Time               : ≥ 08:00
                        :environment       :Time               : ≤ 18:00
               Function: string-equal
                /* :Attribute-Category   :Attribute ID
                       : access-subject     :SubjectName
                       : resource             :FilerName
            </Condition>
        </Rule>
```

```
</Policy>
```

3.5 XACML Access Request

An XACML access request is specified in terms of one or more attributes associated with elements: subject, action, resource, and environment. For example, if the IRS Agent Smith is making a request to write Brown's Tax Return at 9:30 a.m., the XACML access request will carry the values "smith" and "IRS-agent" for the Subject-id and Role attributes, value "write" for action's Action-id, values "tax-return" and "brown" for the resource's Resource-id, and Resource-owner attributes, and value "09:30 a.m." for environment's Time attribute. XACML pseudocode for this access request is as follows.

```
<Request REQ1>
    <Attributes>  /* :Attribute-Category   : Attribute ID   : Attribute Value */
        :access-subject   :Subject-id   :smith
        :access-subject   :Role   :IRS agent
        :resource   :Resource-id   :tax-return
        :resource   :Resource-owner   :brown
        :action   :Action-id   :write
        :environment   :Time   :9:30 a.m.
    </Attributes>
</Request REQ1>
```

3.6 Delegation

The XACML Policies discussed thus far have pertained to Access Policies that are created and may be modified by an authorized administrator. Access Policies specify capabilities for subjects to perform actions on resource objects. An Access Policy is always considered trusted and its authority is not verified by PDP. XACML includes a delegation mechanism to support decentralized administration of a subset of access policies. A consequence of this feature is a new type of policy called an Untrusted Access Policy that must have its authority verified.

In addition to Untrusted Access Policies, the delegation approach makes use of Trusted Administrative Policies and Untrusted Administrative Policies. Administrative policies (trusted or untrusted) include a delegate and a situation in its Target. A *situation* is a means of scoping the access rights that can be delegated and may include some combination of subject, resource, and action attributes. The *delegate* is an attribute category of the same type as subject, thus representing the entity(s) that has been given the authority to create either access or further delegation rights.

Trusted Administrative Policies serve as a root of trust. They are created under the same authority that is used to create Access Policies. A Trusted Administrative Policy gives the delegate the authority to create Untrusted Administrative Policies or Untrusted Access Policies. The situation for a created Untrusted Administrative Policy or Untrusted Access Policy needs to be either the same situation (the same scope) as that of the Trusted Administrative Policy or a subset of the situation (narrower in scope). In addition, an Untrusted Administrative Policy or

Untrusted Access Policy includes a *policy issuer* tag with a value that is the same as the value of the delegate in the administrative policy under which it was created. An Untrusted Administrative Policy provides authority to the delegate to create either: (a) an Untrusted Administrative Policy with a policy issuer, delegate, and situation, or (b) an Untrusted Access Policy with a policy issuer and situation. Both these policies should have at least one rule with a PERMIT or DENY effect.

XACML recognizes two types of requests – Access Requests and Administrative Requests. Access Requests are issued to (attempt to match targets of) Access Policies or Untrusted Access Policies. An Untrusted Access Policy includes a Policy Issuer tag and an Access Policy does not. If the Access Request matches the target of an Access Policy, the PDP considers the Access Policy applicable and it is directly used by PDP in a combining algorithm to arrive at a final decision. If the Access Request matches the target of an Untrusted Access Policy, the authority of the policy issuer must first be verified before it can be considered by the PDP. Authority is determined through establishment of a *delegation chain* from the Untrusted Access Policy, through potentially zero or more Untrusted Administrative Policies, to a Trusted Administrative Policy. If the authority of the policy issuer can be verified, the PDP evaluates the access request against the Untrusted Access Policy; otherwise it is considered an unauthorized policy and discarded. In a graph where policies are nodes, a delegation chain consists of a series of edges from the node representing an Untrusted Access Policy to a Trusted Administrative Policy. To construct each edge of the graph, the XACML context handler formulates Administrative Requests.

An Administrative Request has the same structure as an Access Request except that in addition to attribute categories – access-subject, resource, and action – it also uses two additional attribute categories, delegate and decision-info. If a policy Px happens to be one of the applicable (matched) Untrusted Access Policies, the administrative request is generated using policy Px to construct an edge to policy Py using the following:

- Convert all Attributes (and attribute values) used in the original Access Request to attributes of category delegated.
- Include the value under the *PolicyIssuer* tag of Px as value for the subject-id attribute of the *delegate* attribute category.
- Include the effect of evaluating policy Px as attribute value (PERMIT, DENY, etc.) for the Decision attribute of *decision-info* attribute category.

The Administrative Request constructed using the above attributes is evaluated against the target for policy Py. If the result of the evaluation is "PERMIT", an edge is constructed between policies Px and Py. The overall logic involved is to verify the authority for issuance of policy Px. For this there should exist a policy with its "delegate" set to the policy issuer of Px. If that policy is Py, then it means policy Px has been issued under the authority found in policy Py. The edge construction then proceeds from policy Py until an edge to a Trusted Administrative Policy is found.

The process of selecting applicable policies for inclusion in the combining algorithm is illustrated in Figure 3. Based on the matching of the attributes in the original access request to the targets in various policies, Untrusted Access Policies P31, P32, and P33 can be found to be applicable policies. A path to a Trusted Administrative Policy P11 can be found directly from the applicable Untrusted Access Policy P31. A path to a Trusted Administrative Policy P12 can be found through Untrusted Administrative Policy P22 for the applicable Untrusted Access Policy P32. Because no such path can be found for the third applicable Untrusted Access Policy P33, only policies P31 and P32 will be used in the combining algorithm for evaluating the final access decision, and policy P33 will be discarded since its authority could not be verified.

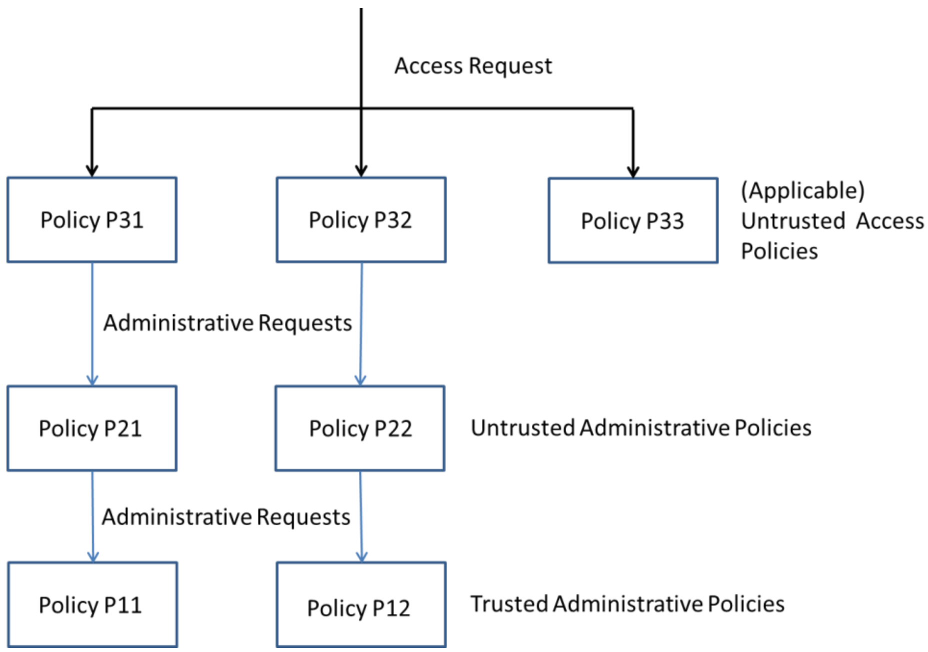

Figure 3: Utilizing Delegation Chains for Policy Evaluation

Below is a more concrete example that illustrates the use of delegation chains to select applicable policies that are used in combining algorithms for arriving at final access decisions. The example gives a Policy Set that consists of four policies:

- Policy P1: A Trusted Administrative Policy that gives John (the delegate) the authority to create policies for a situation involving reading of medical records to any user who has the role of Doctor.
- Policy P2: An Untrusted Administration Policy that is issued by John, under the authority of P1, to give Jessica (the delegate) the authority to create policies for a situation involving reading of medical records to any user who has the role of Doctor. Because of the matching of delegate of P1 to policy issuer of P2 and the fact that the situations in both policies P1 and P2 are the same, it is obvious that the authority to issue policy P2 has come from policy P1. Thus P1 and P2 form a delegation chain.
- Policy P3: An Untrusted Access Policy that is issued by Jeff to give Carol the capability to read medical records.
- Policy P4: An Untrusted Access Policy that is issued by Jessica to give Carol the ability to read medical records. Because of the matching of delegate of P2 to policy issuer of P4 and the fact that the situations in both policies P2 and P4 are the same, it is obvious that

the authority to issue policy P4 has come from policy P2. Thus P2 and P4 form a delegation chain.

The four policies described above are given in the form of pseudocode below:

```
<Policy Set>
  <Policy P1> /* Trusted Administrative Policy */
     <Target>  /* :Attribute-Category  :Attribute ID  :Attribute Value */
       :access-subject  :role  :doctor
       :resource   :resource-id   :medical-records
       :action   :action-id   :read
       :delegate   :subject-id   :john
    </Target>
    <Rule R1>
        Effect:  PERMIT
     </Rule R1>
 </Policy P1>

 <Policy P2> /* Untrusted Administrative Policy */
     <Policy Issuer> john </Policy Issuer>
     <Target>  /* :Attribute-Category   : Attribute ID   : Attribute Value */
         :access-subject   :role   :doctor
         :resource   :resource-id   :medical-records
         :action   :action-id   :read
         :delegate   :subject-id   :jessica
    </Target>
    <Rule R2>
        Effect:  PERMIT
     </Rule R2>
  </Policy P2>

  <Policy P3> /* UnTrusted Access Policy */
     <Policy Issuer> Jeff </Policy Issuer>
     <Target>  /* :Attribute-Category   : Attribute ID   : Attribute Value */
         :access-subject   :subject-id   :carol
         :resource   : resource-id   :medical-records
         :action   :action-id   :read
     </Target>
     <Rule R3>
        Effect:  PERMIT
     </Rule R3>
  </Policy P3>

  <Policy P4> /*  UnTrusted Access Policy */
     <Policy Issuer> Jessica  </Policy Issuer>
     <Target>  /* :Attribute-Category   : Attribute ID   : Attribute Value */
```

```
            :access-subject   :subject-id   :carol
            :resource   :resource-id   :medical-records
            :action   :action-id   :read
        </Target>
       <Rule R4>
          Effect:  PERMIT
        </Rule R4>
    </Policy P4>
<Policy Set>
```

By matching the situation and delegate in one policy to situation and policy issuer in another, we see that P1, P2, and P4 form a delegation chain. P3 is not part of any delegation chain. Given the above delegation structure, let us see how the following access request REQ1 will be resolved.

```
<Request REQ1>
        <Attributes>  /* :Attribute-Category   : Attribute ID   : Attribute Value */
            :access-subject   :subject-id   :carol
            :access-subject   :role   :doctor
            :resource   :resource-id   :medical-records
            :action   :action-id   :read
        </Attributes>
</Request REQ1>
```

By matching the attributes (and values) in the request REQ1 with the attributes (and values) in the target of the policies in the policy set, we find that only policies P3 and P4 match directly since policies P1 and P2 contain delegated attributes. Since both policies P3 and P4 are untrusted access policies, their respective authority has to be verified by making administrative requests. Since policy P3 is not part of any delegation chain, its authority cannot be verified. However, the authority for policy P4 can be established by using the delegation chain P1, P2, P4.

The same PAP interface that is used to create access policies can be used to create the additional policies needed for supporting delegation – Untrusted Access Policies, Trusted Administrative Policies, and Untrusted Administrative Policies. This requires at least two classes of policy administrators. The first is a System-Administrator authorized to create Access Policies. The second is a Delegated-Administrator authorized to create Untrusted Administrative Policies or Untrusted Access Policies conforming to the situation or a subset of the situation authorized in any Trusted Administrative Policy currently in the policy repository.

3.7 XACML Reference Architecture

XACML reference architecture defines necessary functional components (depicted in Figure 4) to achieve enforcement of its policies. The authorization process is a seven-step process that depends on four layers of functionality: Enforcement, Decision, Access Control Data, and Administration.

At its core is a PDP that computes decisions to permit or deny subject requests (to perform actions on resources). Requests are issued from, and PDP decisions are returned to, a PEP using a standardized request and response language. The PEP is implemented as a component of an operating environment that is tightly coupled with its application. A PEP may not generate requests in XACML syntax nor process XACML syntax-compliant responses. In order to convert access requests in native format (of the operating environment) to XACML access requests (or convert a PDP response in XACML to a native format), the XACML architecture includes a context handler. The context handler also provides additional attribute values for the access request context (retrieving them from PIP). In the reference architecture in Figure 4, the context handler is not explicitly shown as a component since we assume that it is an integral part of the PEP or PDP.

A request is comprised of attributes extracted from the PIP, minimally sufficient for Target matching. The PIP is shown as one logical store, but in fact may comprise multiple physical stores. In computing a decision, the PDP queries policies stored in a PRP. If the attributes of the request are not sufficient for rule and policy evaluation, the PDP may request the context handler to search the PIP for additional attributes. Information and data stored in the PIP and PRP comprise the access control data and collectively define the current authorization state.

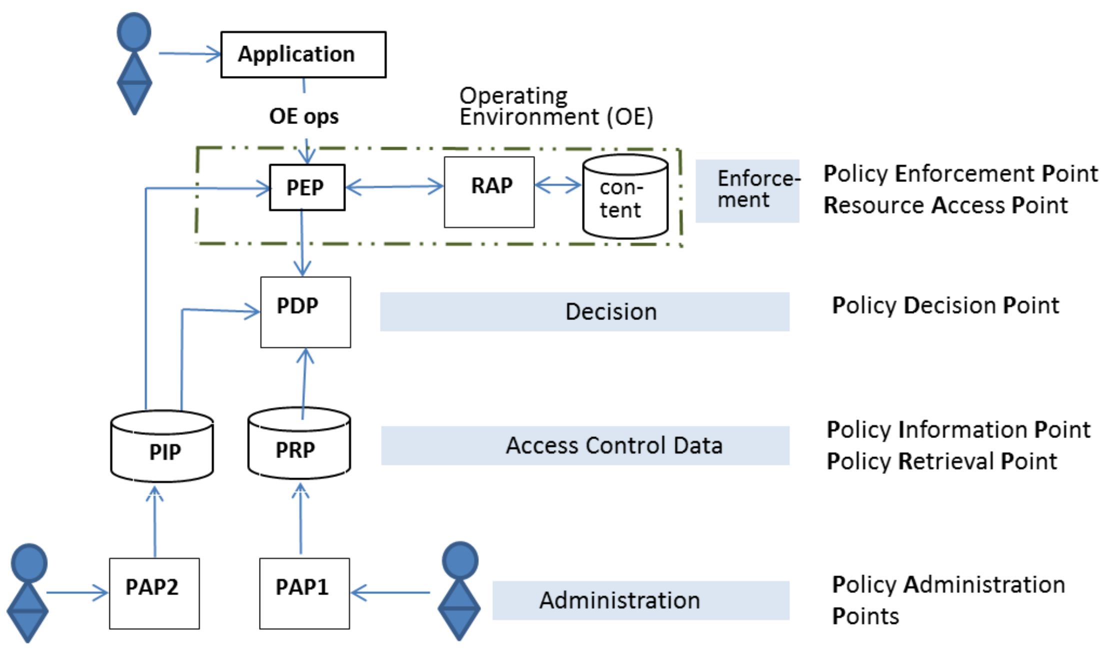

Figure 4: XACML Reference Architecture

A Policy Administration Point (PAP1) using the XACML policy language creates the access control data stored in the PRP in terms of rules for specifying Policies, PolicySets as a container of Policies, and rule and policy combining algorithms. The PRP may store trusted or untrusted policies. Although not included in the XACML reference architecture, we show a second Policy Administration Point (PAP2) for creating and managing the access control data stored in the PIP. PAP2 implements administrative routines necessary for the creation and management of attribute names and values for users and resources. The Resource Access Point (RAP) implements routines for performing operations on a resource that is appropriate for the resource type. In the event that the PDP returns a permit decision, the PEP issues a command to the RAP for execution of an operation on resource content. As indicated by the dashed box in Figure 4, the RAP, in addition to the PEP, runs in an application's operating environment, independent of the PDP and its supporting components. The PDP and its supporting components are typically implemented as modules of a centralized Authorization Server that provides authorization services for multiple types of operations.

4 NGAC Specification

NGAC takes a fundamentally different approach from XACML for representing requests, expressing and administering policies, representing and administering attributes, and computing and enforcing decisions. NGAC is defined in terms of a standardized and generic set of relations and functions that are reusable in the expression and enforcement of policies.

For purposes of brevity and readability, the NGAC specification is presented as a summary that highlights NGAC's salient features and should not be considered complete. In some instances, actual NGAC relational details and terms are substituted with others to accommodate a simpler presentation.

4.1 Basic Policy and Attribute Elements

NGAC's access control data is comprised of basic elements, containers, and configurable relations. While XACML uses the terms subject, action, and resource, NGAC uses the terms user, operation, and object with similar meanings. In addition to these, NGAC includes processes, administrative operations, and policy classes. Like XACML, NGAC recognizes user and object attributes; however, it treats attributes along with policy class entities as containers. These containers are instrumental in both formulating and administering policies and attributes.

NGAC treats users and processes as independent but related entities. NGAC processes can be thought of as simple representations of operating system processes. They have an id, memory, and descriptors for resource allocations (e.g., "handles"). Like an operating system, an NGAC process can utilize system resources (e.g., clipboard) for inter-process communication. Processes through which a user attempts access take on the same attributes as the invoking user.

Although an XACML resource is similar to an NGAC object, NGAC uses the term object as an indirect references its data content. Every object is also an object attribute with the same name. Given this one-to-one correspondence, the object can also be identified as an object attribute. That is, every object is by definition an object attribute. The set of objects reflects entities needing protection, such as files, clipboards, email messages, and record fields.

Similar to an XACML subject attribute value, NGAC user containers can represent roles, affiliations, or other common characteristics pertinent to policy, such as security clearances.

Object containers (attributes) characterize data and other resources by identifying collections of objects, such as those associated with certain projects, applications, or security classifications. Object containers can also represent compound objects, such as folders, inboxes, table columns, or rows, to satisfy the requirements of different data services. Policy class containers are used to group and characterize collections of policy or data services at a broad level, with each container representing a distinct set of related policy elements. Every user, user attribute, and object attribute must be contained in at least one policy class. Policy classes can be mutually exclusive or overlap to various degrees to meet a wide range of policy requirements.

NGAC recognizes a generic set of operations that include basic input and output operations (i.e., read and write) that can be performed on the contents of objects that represent data service resources, and a standard set of administrative operations that can be performed on NGAC access control data that represent policies and attributes. In addition, an NGAC deployment may consider and provide control over other types of data service operations besides the basic input/output operations. Resource operations can also be defined specifically for an operating environment. Administrative operations, on the other hand, pertain only to the creation and deletion of NGAC data elements and relations, and are a stable part of the NGAC framework, regardless of the operating environment.

4.2 Relations

NGAC does not express policies through rules, but instead through configurations of relations of four types: assignments (define membership in containers), associations (derive privileges), prohibitions (specify privilege exceptions), and obligations (dynamically alter access state).

4.2.1 Assignments and Associations

NGAC uses a tuple (x, y) to specify the assignment of element x to element y. In this publication we use the notation x→y to denote the same assignment relation. The assignment relation always implies containment (x is contained in y). We denote a chain of one or more assignment relations by “$\rightarrow^{+}$”.The set of entities used in assignments include users, user attributes, and object attributes (which include all objects), and policy classes.

To be able to carry out an operation, one or more access rights are required. As with operations, two types of access rights apply: non-administrative and administrative.

Access rights to perform operations are acquired through associations. An association is a triple, denoted by *ua---ars---at*, where *ua* is a user attribute, *ars* is a set of access rights, and *at* is an attribute, where *at* may comprise either a user attribute or an object attribute. The attribute *at* in an association is used as a referent for itself and the policy elements contained by the attribute. Similarly, the first term of the association, attribute *ua*, is treated as a referent for the users and user attributes contained in *ua*. The meaning of the association *ua---ars---at* is that the users contained in *ua* can execute the access rights in *ars* on the policy elements referenced by *at*. The set of policy elements referenced by *at* is dependent on (and meaningful to) the access rights in *ars*.

Figure 5 illustrates two example assignment and association relations depicted as graphs—one an access control policy configuration with policy class “Project Access” (Figure 5a), and the other a data service configuration with “File Management” as its policy class (Figure 5b). Users and user attributes are on the left side of the graphs, and objects and object attributes are on the right. The arrows represent assignment relations and the dashed lines denote associations. Remember that the set of referenced policy elements is dependent on the access rights in ars. Note that the *at* attribute of each association is an object attribute and the access rights are read/write. In the association Division---{r}---Projects, the policy elements referenced by Projects are objects o1 and o2, meaning that users u1 and u2 can read objects o1 and o2. If we had an association Division---{create assign-to}---Projects, then the policy elements referenced by Projects would be Projects, Project1, and Project2, meaning that users u1 and u2 may (administratively) create assignment relations to Projects, Project1, and Project2.

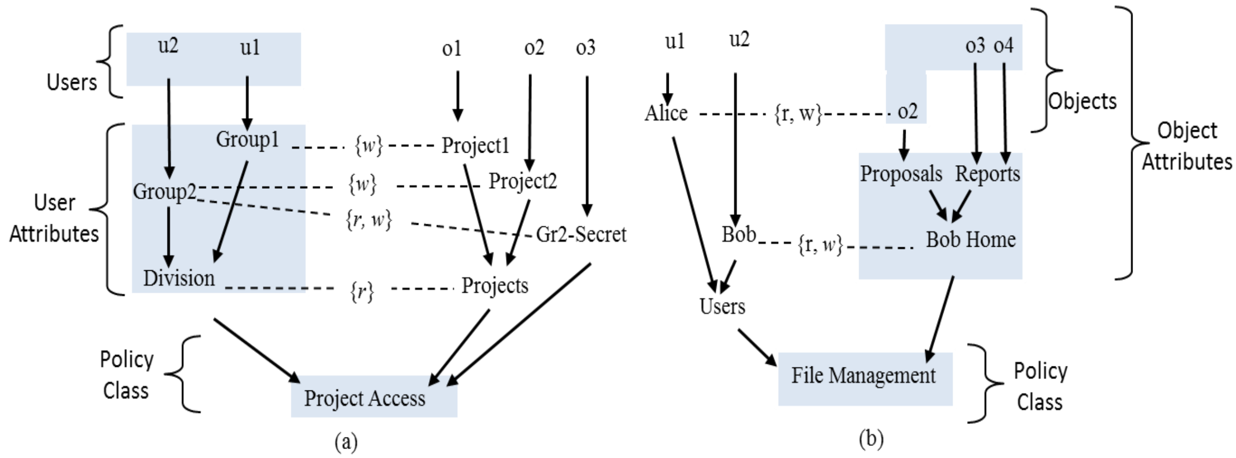

Figure 5: Two Example Assignment and Association Graphs

4.2.2 Derived Privileges

Collectively associations and assignments indirectly specify privileges of the form (*u*, *ar*, *e*), with the meaning that user *u* is permitted (or has a capability) to execute the access right *ar* on element *e*, where *e* can represent a user, user attribute, or object attribute. Determining the existence of a privilege (a derived relation) is a requirement of, but as we discuss later, not sufficient in computing an access decision.

NGAC includes an algorithm for determining privileges with respect to one or more policy classes and associations. Specifically, (*u*, *ar*, *e*) is a privilege, if and only if, for each policy class *pc* in which *e* is contained, the following is true:

- The user *u* is contained by the user attribute of an association;
- The element *e* is contained by the policy element of that association;
- The policy element of that association is contained by the policy class *pc*, and
- The access right *ar* is a member of the access right set of that association.

Note that the algorithm for determining privileges applies to configurations that include one or more policy classes. The left and right columns of Table 2 list derived privileges for Figures 5a and 5b, when considered independent of one another.

Table 2: Derived Privileges for the Independent Configuration of Figures 5a and 5b

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3)	(u1, r, o2), (u1, w, o2), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Figure 6 is an illustration of the graphs in Figures 5a and 5b when considered in combination. Note that for the purposes of deriving privileges, user attribute to policy class assignments are not considered, and as such are not shown.

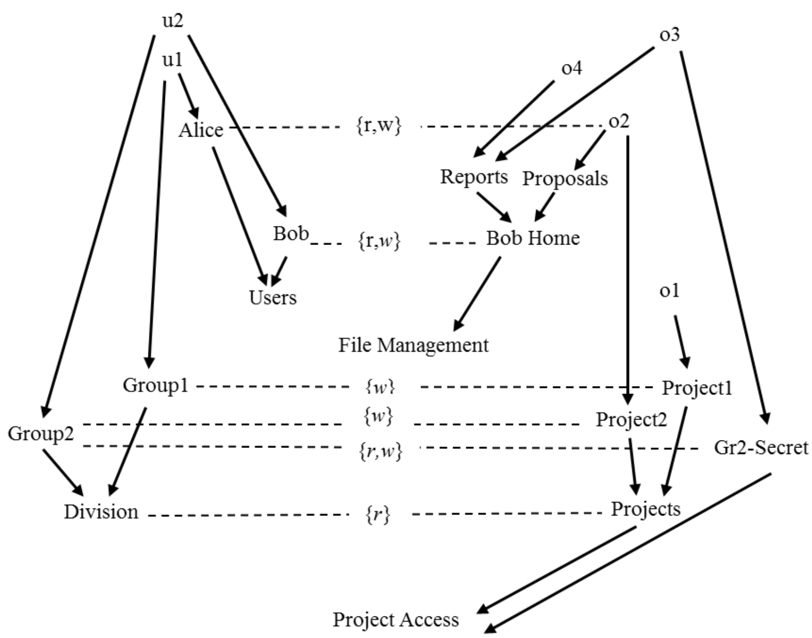

Figure 6: Graphs from Figures 5a and 5b in Combination

Table 3 lists the derived privileges for the graphs from Figures 5a and 5b when considered in combination.

Table 3: Derived Privileges for the Combined Configuration of Figures 5a and 5b

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Note that (u1, r, o1) is a privilege in Table 2 because o1 is only in policy class Project Access and there exists an association Division---{r}--- Projects, where u1 is in Division, r is in {r}, and o1 is in Projects. Note that (u1, w, o2) is not a privilege in Table 2 because o2 is in both Project Access and File Management policy classes, and although there exists an association Alice---{r, w}---o2, where u1 is in Alice, w is in {r, w}, and o2 is in o2 and File Management, no such association exists with respect to Project Access.

NGAC configurations indirectly specify rules. The access control policy of Figure 5a specifies that users assigned to either Group1 or Group2 can read objects contained in Projects, but only Group1 users can write to Project1 objects and only Group2 users can write to Project2 objects. The Policy further specifies that Group2 users can read/write data objects in Gr2-Secret. While Figure 5a specifies policies for how its objects can be read and written, the configuration is considered incomplete in that it does not specify how its users, objects, policy elements, assignments, and associations were created and can be managed.

Figure 5b depicts an access policy for a File Management data service. User u2 (Bob) has read/write access to objects assigned to object attributes (Proposals and Reports representing folders) that are contained in Bob Home (representing his home directory). The configuration also shows user u1 (Alice) with read/write access to object o2. This configuration is also incomplete in that one would expect a File Management data service with capabilities for users to create and manage their folders and to create and assign objects to their folders. Another feature common to a File Management data service is the capability for users to grant or give away access rights to objects that are under their control to other users.

We specify missing management capabilities for the Project Access policy in Section 4.4.1 and File Management data service in Section 4.5.

Although the graph depicted in figure 6 pertains to the intersection of policies, NGAC employs the Boolean logics of AND and OR to express the combinations of policies [12]. Figure 7 is a depiction of an NGAC equivalent configuration of the XACML Policy1 specified in Section 3.4. Both policies specify that users assigned to Intern can read AND Doctor can read and write Medical Records that are assigned to the same Ward as the user OR Doctors can read and write Medical Records assigned to Critical regardless of the Ward in which the Medical Record is assigned.

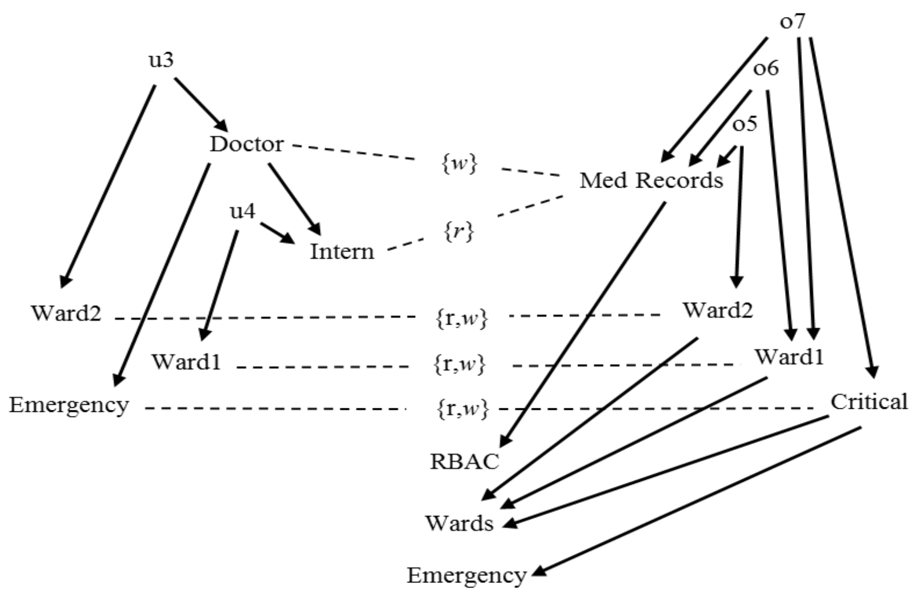

Figure 7: NGAC's Equivalent Expression of XACML Policy1

Figure 7 shows NGAC users and objects that correspond to the XACML subjects and resources in Table 1 and are assigned to the same attribute values in Table 1.

Table 4: Derived Privileges for the Configuration of Figure 7

(u3, r, o5), (u3, w, o5), (u3, r, o7), (u3, w, o7), (u4, r, o6)

As a consequence, the derived privileges of Figure 7 (listed in Table 4) are the same as the authorization state specified in Table 1.

4.2.3 Prohibitions (Denies)

In addition to assignments and associations, NGAC includes three types of prohibition relations: user-deny, user attribute-deny, and process-deny. In general, deny relations specify privilege exceptions. We respectively denote a user-based deny, user attribute-based deny, and process-based deny relation by u_deny(*u*, *ars*, *pe*), ua_deny(*ua*, *ars*, *pe*), and p_deny(*p*, *ars*, *pe*), where *u* is a user, ua is a user attribute, *p* is a process, *ars* is an access right set, and *pe* is a policy element used as a referent for itself and the policy elements contained by the policy element. The respective meanings of these relations are that user *u*, users in *ua*, and process *p* cannot execute access rights in *ars* on policy elements in *pe*. User-deny relations and user attribute-deny relations can be created directly by an administrator or dynamically as a consequence of an obligation (see Section 4.2.4). An administrator, for example, could impose a condition where no user is able to alter their own Tax Return, in spite of the fact that the user is assigned to an IRS Auditor user attribute with capabilities to read/write all tax returns. When created through an obligation, user-deny and user attribute-deny relations can take on dynamic policy conditions. Such conditions can, for example, provide support for separation of duty policies (if a user executed capability x, that user would be immediately precluded from being able to perform capability y). In addition, the policy element component of each prohibition relation can be specified as its complement, denoted by ¬. The respective meaning of u_deny(*u*, *ars*, ¬*pe*), ua_deny(*ua*, *ars*, ¬*pe*), and p_deny(*p*, *ars*, ¬*pe*) is that the user *u*, and any user assigned to *ua*, and process *p* cannot execute the access rights in *ars* on policy elements not in *pe*.

Process-deny relations are exclusively created using obligations. Their primary use is in the enforcement of confinement conditions (e.g., if a process reads Top Secret data, preclude that process from writing to any object not in Top Secret).

4.2.4 Obligations

Obligations consist of a pair (*ep*, *r*) (usually expressed as **when** *ep* **do** *r*) where *ep* is an *event pattern* and *r* is a sequence of administrative operations, called a *response*. The event pattern specifies conditions that if matched by the context surrounding a process's successful execution of an operation on an object (an event), cause the administrative operations of the associated response to be immediately executed. The context may pertain to and the event pattern may specify parameters like the user of the process, the operation executed, and the attribute(s) of the object.

Obligations can specify operational conditions in support of history-based policies and data services. Such conditions include conflict of interest (if a user reads information from a sensitive data set, that user is prohibited from reading data from a second data set) and Work Flow (approving (writing to a field of)) a work item enables a second user to read and approve the work item). Also, included among history-based policies are those that prevent leakage of data to unauthorized principals. The use of an obligation to prevent data leakage is discussed in Section 4.5.

4.3 NGAC Decision Function

The NGAC access decision function controls accesses in terms of processes. The user on whose behalf the process operates must hold sufficient authority over the policy elements involved. The function process_user(*p*) denotes the user associated with process *p*.

Access requests are of the form (*p*, *op*, *argseq*), where *p* is a process, *op* is an operation, and *argseq* is a sequence of one or more arguments, which is compatible with the scope of the operation. That is, an access request comprises an operation and a list of enumerated arguments that have their number, type, and order dictated by the operation.

The access decision function to determine whether an access request can be granted requires a mapping from an operation and argument sequence pair to a set of access rights and policy element pairs (i.e., {(*ar*, *pe*)}) the process's user must hold for the request to be granted.

When determining whether to grant or deny an access request, the authorization decision function takes into account all privileges and restrictions (denies) that apply to a user and its processes, which are derived from relevant associations and denies, giving restrictions precedence over privileges:

> A process access request (*p*, *op*, *argseq*) with mapping (*op*, *argseq*)→{(*ar*, *pe*)}) is granted iff for each (ar_i, pe_i) in {(*ar*, *pe*)}, there exists a privilege (*u*, ar_i, pe_i) where *u* = process_user(*p*), and (ar_i, pe_i) is not denied for either *u* or *p*.

In the context of Figure 6, an access request may be (p, read, o1) where p is u1's process. The pair (read, o1) maps to (r, o1). Because there exists a privilege (u1, r, o1) in table 3 and (r, o1) is not denied for u1 or p, the access request would be granted. Assume the existence of associations Division---{create assign-to}---Projects, and Bob---{create assign-from}---Bob Home in the context of Figure 6, and an access request (p, assign, <o4, Project1>) where p is u2's process. The pair (assign, <o4, Project1>) maps to {(create assign-from, o4), (create assign-to, Project1)}. Because privileges (u2, create assign-from, o4) and (u2, create assign-to, Project1) would exist under the assumption, and (create assign-from, o4) and (create assign-to, Project1) are not denied for u2 or p, the request would be granted.

4.4 Administrative Considerations

Many access rights categorized as administrative access rights, such as those needed to create a file and assign it to a folder, arguably seem non-administrative from a usage standpoint. Nevertheless, from a policy specification standpoint, they are considered administrative (e.g., in this case, an association with access rights for creating an object and assigning the object to an object attribute is needed). The main difference between the two types of access rights is that non-administrative actions pertain to activities on protected resources represented as objects, while administrative actions pertain to activities on the policy representation comprising the policy elements and relationships defined within and maintained by NGAC.

4.4.1 Administrative Associations

In order to execute an administrative operation, the requesting user must possess appropriate access rights. Just as access rights to perform read/write operations on resource objects are defined in terms of associations, so too are capabilities to perform administrative operations on policy elements and relations. In comparison with non-administrative access rights, where resource operations are synonymous with the access rights needed to carry out those operations (e.g., a "read" operation corresponding to an "r" access right), the authority associated with an administrative access right is not necessarily synonymous with an administrative operation. Instead, the authority stemming from one or more administrative access rights may be required for a single operation to be authorized.

Some administrative access rights are explicitly divided into two parts, as denoted by the "from" and "to" suffixes. Both parts of the authority must be held to carry out the implied administrative operation.

For example, consider the following two associations that provide administrative capabilities in support of the "Project Access" policy configuration depicted in Figure 5a:

ProjectAccessAdmin --- {create-u-to, delete-u-from, create-ua-to, delete-ua-from, create-uua-from, create-uua-to, delete-uua-from, create-uaua-from, create-uaua-to, delete-uaua-from, delete-uaua-to }---Division

ProjectAccessAdmin --- {create-o-to, delete-o-from, create-oa-to, delete-oa-to, create ooa-from, create ooa-to, delete-ooa-from, create-oaoa-from, create-oaoa-to, delete-oaoa-from, delete-oaoa-to }--- Projects

The meaning of the first association is that users in ProjectAccessAdmin can create and delete users, user attributes, user to user-attribute (uua), and user-attribute to user-attribute (uaua) assignments in Division. The second association similarly establishes privileges to create and delete objects(o), object attributes(oa), object to object-attribute (ooa), and object-attribute to object-attribute (oaoa) assignments in Projects.

With the preceding two associations, the next two associations complete the configuration begun by the configuration of Figure 5a, enabling complete administration. The associations enable users in ProjectAccessAdmin to create and delete associations from user attributes in Division to object attributes in Projects, with allocated read and/or write access rights.

ProjectAccessAdmin --- {create-assoc-from, delete-assoc-from} --- Division.
ProjectAccessAdmin --- {create-assoc-to, delete-assoc-to, r-allocate, w-allocate} --- Projects.

4.4.2 Delegation

The question remains, how are administrative capabilities created? The answer begins with a superuser with capabilities to perform all administrative operations on all access control data. The initial state consists of an NGAC configuration with empty data elements, attributes, and relations. A superuser either can directly create administrative capabilities or more practically can create administrators and delegate to them capabilities to create and delete administrative privileges. Delegation and rescinding of administrative capabilities is achieved through creating and deleting associations. The principle followed for allocating access rights via an association is that the creator of the association must have been allocated the access right over the attribute in question (as well as the necessary create-assoc-from and create-assoc-to rights) in order to delegate them. The strategy enables a systematic approach to the creation of administrative attributes and delegation of administrative capabilities, beginning with a superuser and ending with users with administrative and data service capabilities.

4.4.3 NGAC Administrative Commands and Routines

Administrative commands and routines are the means by which policy specifications are formed. Each access request involving an administrative operation corresponds on a one-to-one basis to an administrative routine, which uses the sequence of arguments in the access request to perform the access. As described earlier in this section, the access decision function grants the access request (and initiation of the respective administrative routine) only if the process holds all prohibition-free access rights over the items in the argument sequence needed to carry out the access. The administrative routine, in turn, uses one or more administrative commands to perform the access.

Administrative commands describe rudimentary operations that alter the policy elements and relationships of NGAC, which comprise the authorization state. An administrative command is represented as a parameterized procedure, with a body that describes state changes to policy that occur when the described behavior is carried out (e.g., a policy element or relation Y changes state to Y′ when some function f is applied). Administrative commands are specified using the following format:

cmdname (x_1: type$_1$, x_2: type$_2$, …, x_k: type$_k$)
…*preconditions* …
{
$Y' = f(Y, x_1, x_2, \ldots, x_k)$
}

Consider, as an example, the administrative command CreateAssoc shown below, which specifies the creation of an association. The preconditions here stipulate membership of the x, y, and z parameters respectively to the user attributes (UA), access right sets (ARs), and policy elements (PE) elements of the model. The body describes the addition of the tuple (x, y, z) to the set of associations (ASSOC) relation, which changes the state of the relation to ASSOC′.

createAssoc (x, y, z)
$x \in UA \land y \in ARs \land z \in PE \land (x, y, z) \notin ASSOC$
{
$ASSOC' = ASSOC \cup \{(x, y, z)\}$
}

Each administrative command entails a modification to the NGAC configuration that involves the creation or deletion of a policy element, the creation or deletion of an assignment between policy elements, or the creation or deletion of an association, prohibition, or obligation.

Compared to administrative routines, administrative commands are elementary. That is, administrative commands provide the foundation for the NGAC framework, while administrative routines use one or more administrative commands to carry out their function.

An administrative routine consists mainly of a parameterized interface and a sequence of administrative command invocations. Administrative routines build upon administrative commands to define the protection capabilities of the NGAC model. The body of an administrative routine is executed as an atomic transaction—an error or lack of capabilities that causes any of the constituent commands to fail execution causes the entire routine to fail, producing the same effect as though none of the commands were ever executed. Administrative routines are specified using the following format:

rtnname (x_1: $type_1$, x_2: $type_2$, …, x_k: $type_k$)
… *preconditions* …
{
cmd_1;
$condition_a$ cmd_2, cmd_3;
. . .
$condition_z$ cmd_n;
}

The name of the administrative routine, rtnname, precedes the routine's declaration of formal parameters, x1: type1, x2: type2, …, xk: typek (k ≥ 0). Each formal parameter of an administrative routine can serve as an argument in any of the administrative command invocations, cmd1, cmd2, …, cmdn (n ≥ 0), that make up the body of the routine, and also in any condition prepended to a command. As with an administrative command, the body of an administrative routine is prefixed by preconditions, which in general ensure that the arguments supplied to the routine are valid, and that certain properties on which the routine relies are maintained. As illustrated above, an optional condition can precede one or more of the commands.

For example, when a new user is created, an administrator typically creates a number of containers, links them together, and grants the authority for the user to access them as its work space. Rather than manually performing each step of this sequence of administrative actions for each new user, the entire sequence of repeated actions can be defined as a single administrative routine and executed in its entirety as an atomic action.

To execute the routine, the user (administrative) must possess the necessary capabilities to execute each administrative command.

4.5 Arbitrary Data Service Operations and Policies

NGAC recognizes administrative operations for the creation and management of its data elements and relations that represent policies and attributes, and basic input and output operations (e.g., read and write) that can be performed on objects that represent data service resources. In accommodating data services, NGAC may establish and provide control over other types of operations, such as send, submit, approve, and create folder. However, it does not necessarily need to do so. This is because the basic data service capabilities to consume, manipulate, manage, and distribute access rights on data can be attained as combinations of read/write operations on data and administrative operations on data elements, attributes, and relations that may alter the access state for which users can read/write data.

Consider the following administrative routine that creates a “file management” user and provides the user with capabilities to create and manage objects and folders, and control and share access to objects in the context of Figure 5b. The routine assumes the pre-existence of the user attribute “Users” assigned to the “File Management” policy class as shown in Figure 5b.

```
create-file-mgmt-user(user-id, user-name, user-home) {
  createUAinUA(user-name, Users);
  createUinUA(user-id, user-name);
  createOAinPC(user-home, File Management);
  createAssoc(user-name, {r, w}, user-home);
  createAssoc(user-name, {create-o-to, delete-o-from}, user-home);
  createAssoc(user-name, {create-ooa-from, create-ooa-to, delete-ooa-from, create-oaoa-
      from, create-oaoa-to, delete-oaoa-from}, user-home);
  createAssoc(user-name, {create-assoc-from, delete-assoc-from}, Users);
  createAssoc(user-name, {create-assoc-to, delete-assoc-to, r-allocate, w-allocate}, user-
      home);}
```

This routine with parameters (*u*1, *Bob* and *Bob Home*) could have been used to create “file management” data service capabilities for user *u*1 already in Figure 5b. Through the routine the user attribute “Bob” is created and assigned to “Users”, and user *u*1 is created and assigned to “Bob”. In addition, the object attribute “Bob Home” is created and assigned to policy class “File Management”. In addition, user *u*1 is delegated administrative capabilities to create, organize, and delete object attributes (presented folders) in Bob Home, and *u*1 is provided with capabilities to create, read, write, and delete objects that correspond to files and place those files into his folders. Finally, *u*1 is provided with discretionary capabilities to “grant” to other users in the “Users” container capabilities to perform read/write operations on individual files or to all files in a folder in his Home.

As already indicated by Figure 5b, and subsequent to the execution of this administrative routine, user *u*1 can grant user *u*2 (Alice) read/write access to object *o*2 by using the following routine.

```
grant(user-name, rights, file/folder) {
  createAssoc(user-name, rights, file/folder)}
```

Through this routine Bob could, under his discretion, “grant” Alice read access to o3. However, even if Bob were to do so, Alice would not be able to read o3. This is because of a lack of a privilege (u1, r, o3) due to o3’s containment in the “Project Access” policy class. Although Bob cannot successfully provide Alice read access to object o3 through his delegated “grant” capability, Bob could “leak” the capability to read the content of o3 to Alice. This could be achieved by Bob first reading the content of o3 and then writing that content to o2. Even if Bob was trusted not to perform such actions, a malicious process acting on Bob’s behalf could do so, without Bob's knowledge. To prevent this leakage we add the following obligation to our configuration:

When any process *p* performs (*r*, *o*) where $o \rightarrow^{+}$ Gr2-Secret **do** create p-deny(*p*, {*w*}, ¬Gr2-Secret)

The effect of this obligation will prevent a process (and its user) from reading the contents of any object in Gr2-Secret and writing it to an object in a different container (not in Gr2-Secret).

4.6 NGAC Functional Architecture

NGAC's functional architecture (shown in Figure **Error! Reference source not found.**8), like XACML's, encompasses four layers of functional decomposition: Enforcement, Decision, Administration, and Access Control Data, and involves several components that work together to bring about policy-preserving access and data services. Among these components is a PEP that traps application requests. An access request includes a process id, user id, operation, and a sequence of one or more operands mandated by the operation that pertain to either a data resource or an access control data element or relation. Administrative operational routines are implemented in the PAP and read/write routines are implemented in the RAP.

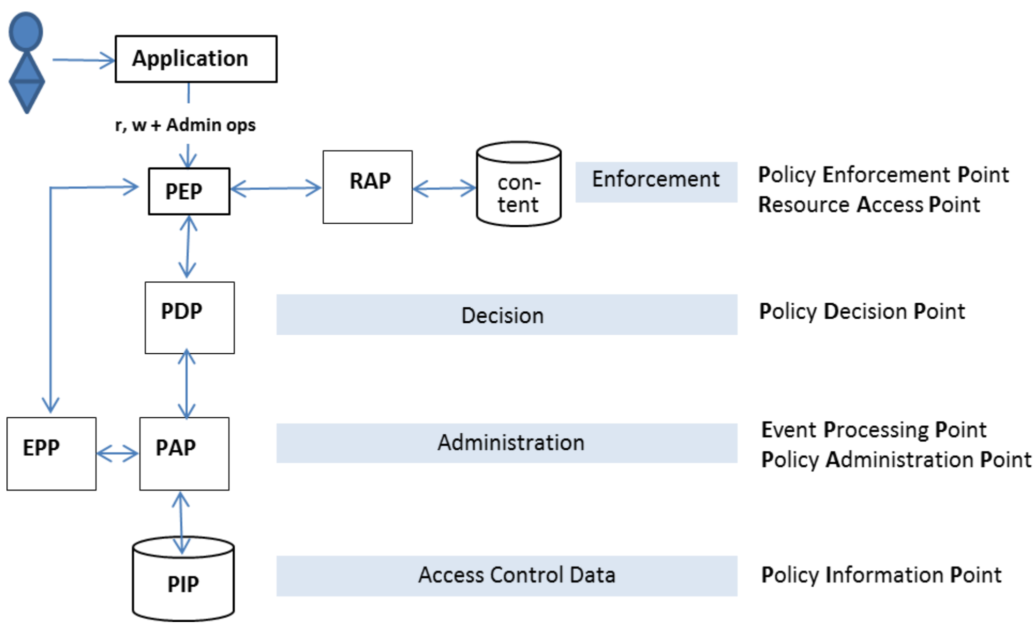

Figure 8: NGAC Standard Functional Architecture

To determine whether to grant or deny, the PEP submits the request to a PDP. The PDP computes a decision based on current configuration of data elements and relations stored in the PIP, via the PAP. Unlike the XACML architecture, the access request information from an NGAC PEP together with the NGAC relations (retrieved by the PDP) provide the full context for arriving at a decision. The PDP returns a decision of grant or deny to the PEP. If access is granted and the operation was read/write, the PDP also returns the physical location where the object's content resides, the PEP issues a command to the appropriate RAP to execute the operation on the content, and the RAP returns the status. In the case of a read operation, the RAP also returns the data type of the content (e.g., Powerpoint) and the PEP invokes the correct data service application for its consumption. If the request pertained to an administrative operation and the decision was grant, the PDP issues a command to the PAP for execution of the operation on the data element or relation stored in the PIP, and the PAP returns the status to the PDP, which in turn relays the status to the PEP. If the returned status by either the RAP or PAP is "successful", the PEP submits the context of the access to the Event Processing Point (EPP). If the context matches an event pattern of an obligation, the EPP automatically executes the administrative operations of that obligation, potentially changing the access state. Note that NGAC is data type agnostic. It perceives accessible entities as either data or access control data elements or relations, and it is not until after the access process is completed that the actual type of the data matters to the application.

5 Analysis

XACML is similar to NGAC insofar as they both provide flexible, mechanism-independent representations of policy rules that may vary in granularity, and they employ attributes in computing decisions. However, XACML and NGAC differ significantly in their expression of policies, treatment of attributes, computation of decisions, and representation of requests. In this section, we analyze these similarities and differences with respect to the degree of separation of access control logic from proprietary operating environments and four ABAC considerations identified in NIST SP 800-162: operational efficiency, attribute and policy management, scope and type of policy support, and support for administrative review and resource discovery.

For the purposes of comparison we normalize some XACML and NGAC terminology.

5.1 Separation of Access Control Functionality from Proprietary Operating Environments

XACML and NGAC both separate access control functionality of data services from proprietary operating environments, but to different degrees. An XACML deployment may consist of multiple operating environments, each hosting one or more applications and sharing a common authorization infrastructure. Each of these operating environments implements its own method of authentication, and in support of its applications implements its own operational routines. Application specific operations included in XACML access requests correspond one-to-one with operational routines implemented in supporting operating environments. It is for this reason that an XACML-enabled application is dependent on an operating environment PEP. Requests are issued from, and decisions are returned to, an operating environment-specific PEP.

Although an NGAC deployment could include a PEP with an Application Programming Interface (API) that recognizes operating environment-specific operations (e.g., send and forward operations for a messaging system), it does not necessarily need to do so. NGAC includes a PEP with an API that supports a set of generic, operating environment-agnostic operations (read, write, create, and delete policy elements and relations). This API enables a common, centralized PEP to be implemented to serve the requests of multiple applications. Although the generic operations may not meet the requirements of every application (e.g., transactions that perform computations on attribute values), calls from many applications can be accommodated. This includes operations that generically pertain to consumption, manipulation, and management of data, and distribution of access rights on data. For example, the “send” operation of a messaging data service could be implemented through a series of administrative operations on NGAC data elements and relations, where “inboxes” and “outboxes” are represented as object attributes. The administrative operations create and assign a message (an object) to the “outbox” of the sender and the “inbox” of the recipient, where the sender and recipient have read access rights to objects contained in their respective “outbox” and “inbox”. The file management data service described in Section 4 is another example of a data service that supports application specific operations for creating and managing files and folders implemented though NGAC generic operations. Still others could include operations in support of workflow, calendar, record management, and time and attendance.

XACML does not envisage the design of a PEP that is data service agnostic. In other words, a PEP under the XACML architecture is tightly coupled to a specific operating environment for which it was designed to enforce access. However, based on the deployment feature described above, it is possible for the NGAC PEP to provide a level of abstraction between application calls and underlying object types and their associated privileges.

As a consequence of this abstraction capability, NGAC can completely displace the need for an access control mechanism of an operating environment in that through the same API, set of operations, access control data elements and relations, and functional components, arbitrary data services can be delivered to users, and arbitrary, mission-tailored access control policies can be expressed and enforced over executions of application calls.

5.2 Scope and Type of Policy Support

Access control policy is a broad term that pertains to many types of controls. For purposes of this report, we subdivide these controls into two broad categories: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In addition, we further categorize MAC into two subcategories, those that support confinement and those that do not.

DAC is an administrative policy that permits system users to allow or disallow other users' access to resources/objects under their control. The means of restricting access to objects is often based on the identities of users and/or the attributes to which they are assigned. The controls are discretionary in the sense that a user with access to a resource is capable of passing that access on to other users without the intercession of a system administrator [15]. Although XACML can theoretically implement DAC policies, it is not efficient. Consider the propagation feature of DAC. DAC permits owners/creators of objects to grant some or all of their capabilities to other users, and the grantees can further propagate those capabilities on to other users. The overall DAC feature to grant privileges to another user and the ability of the grantee to propagate those privileges cannot be supported in XACML syntax using "Access Policies" alone. XACML is geared for specifying global access policies in terms of attributes. Since the only user attribute designator is "access-subject", there is no predefined attribute category to denote the owner/creator of an object.

Therefore, all the capabilities of the owner/creator of an object together with administrative capabilities to grant those privileges have to be specified using a Trusted Administrative policy. The capabilities held by owner/creator can be captured by designating the owner/creator of the object as the "access-subject", and the administrative capability to grant privileges to others can be captured by designating the owner/creator as a delegate in that policy type. The creation of this trusted administrative policy, in turn, enables creation of derived administrative policies with the owner/creator as the policy issuer with the specified set of capabilities. Further, the specification of a "delegate" in this derived administrative policy (labeled NOT TRUSTED) provides a means for the owner/creator to grant capabilities to other users, as well as the ability for the grantee to propagate those capabilities to other users. However, while it is theoretically possible to implement DAC by leveraging XACML's delegation feature, this approach involves significant administrative overhead. The solution requires the specification of a trusted administrative policy and a set of derived administrative policies for every object owner/creator, and for all grantees of the capabilities.

NGAC offers a flexible means of providing users with administrative capabilities to include those necessary for the implementation of different flavors of DAC. As shown by the execution of the administrative routine "create-file-mgmt-user(user-id, user-name, user-home)" in Section 4.5, user $u1$ (Bob) is created and given "File Management" data service capabilities. These capabilities include being able to create objects and assign them to his home, and consequently, having read/write access to those objects. In addition, Bob is given ownership and control capabilities over objects in his home (i.e., Bob can grant other users (e.g., Alice) read/write access to any object in his home). Because Alice is also a "File Management" user, Alice could create a copy of the object, place it in her home, and grant other users access to her copy.

In contrast to DAC, MAC enables ordinary users' capabilities to execute resource operations on resource objects, but not administrative capabilities that may influence those capabilities. MAC policies unavoidably impose rules on users in performing operations on resource objects.

Expression of MAC policies is perhaps XACML's strongest suit. XACML can specify rules in terms of attribute values that can be of varying types, such as strings and integers. There are undoubtedly certain policies that are expressible in terms of these rules that cannot be easily accommodated by NGAC. For example, a financial transaction may pertain to adding a person's credit limit to their account balance. XACML also takes into consideration environmental attributes in expressing policies, and NGAC does not directly support such policies. These environmental-driven policies are dynamic in nature in that the authorization state can change without the involvement of any administrative action. For instance, the threat level can change from "Low" to "High". XACML also includes the notion of an obligation that directs a PEP to take an action prior to or after an access request is approved or denied. XACML obligation can complement and refine MAC policies in a number of ways. While NGAC also uses the term obligation, an NGAC obligation refers to a different policy construct.

MAC policies are often dependent on and include administrative policies. This is especially true in a federated or collaborative environment, where governance policies require different organizational entities to have different responsibilities for administering different aspects of policies and their dependent attributes. It is also often desirable to be able to express policies that prevent combinations of resource capabilities and administrative capabilities—for example, a policy that would prevent an administrator from granting him/herself access to sensitive resources. XACML is ill suited to naturally express such policies. Consider the MAC policy depicted by Figure 5a. Although XACML can certainly express and enforce this policy, it cannot easily express policies as to who can assign users to the various groups (attributes), while NGAC can. NGAC can create administrative attributes and provide users with administrative capabilities down to the granularity of a single configuration element. Furthermore, NGAC can deny administrative capabilities down to the same granularity.

Although XACML has been shown to be capable of expressing aspects of standard RBAC [1] through an XACML profile [16], the profile falls short of demonstrating support for dynamic separation of duty, a key feature used for accommodating the principle of least privilege, and separation of duty, a key feature for combatting fraud. Annex B of Draft standard Next Generation Access Control – Generic Operations and Data Structures (NGAC-GOADS) [20] demonstrates NGAC support for all aspects of the RBAC standard. The appendix also demonstrates support for the Chinese wall policy [4], which cannot be entirely accommodated by XACML.

NGAC has shown support for history-based separation of duty [7]. Simon and Zurko, in their seminal paper on separation of duty [19], describe history-based separation of duty as the most accommodating form of separation of duty, subsuming the policy objectives of other forms. Other history-based policies that can be accommodated by NGAC include two-person control, workflow, and conflict-of-interest.

Despite the use of attributes, the policies discussed thus far have resulted in a user-based authorization state. In other words, the policies and attributes together constitute an authorization state of the form {(*u*, *ar*, *o*)}, where user *u* is authorized to access object *o* under the access right *ar*. Such policies ignore the fact that processes, not users, actually access object content. In general, user-based authorization controls (whether MAC or DAC) share a weakness: their inability to prevent the “leakage” of data to unauthorized principals through malware, or malicious or complacent user actions.

To illustrate this weakness, assume the following authorization state {(u1, r, o1), (u1, w, o2), and (u2, r, o2)}. Note that it is impossible to determine if u2 can read the content of o1. Under one scenario, u1 can read and subsequently write the contents of o1 to o2. Even if policy depended on “trust in users”, we must all but assume the existence of a Trojan horse that can easily thwart policy. This threat exists because, in reality, users do not perform operations on objects, but under a user’s capabilities, processes perform operations (actions) on the content of objects (resources). Therefore, a program executed by u1 can read the contents of o1 and, without u1’s further action or knowledge, write that content to o2. Note that one cannot prevent this leakage even with the addition of a user-based deny condition or relation NOT (u2, r, o1). The importance of preventing inappropriate leakage of data (often called confinement) was recognized as early as the 1970s, with the establishment of the Bell and LaPadula security model [3] and the specific MAC policy defined in Trusted Computer Security Evaluation Criteria (TCSEC) [5].

Because XACML does not allow the specification and enforcement of policies that pertain to processes in isolation of their users, it excludes or imposes undue constraints on users in regard to MAC confinement policies. Another drawback of XACML is that its PDP is stateless, which places limitations on the policies that can be specified and enforced. Although XACML includes the concept of an obligation, it is not used to alter authorization state.

Consider the following XACML TCSEC MAC policy specification:

```
<Policy PolicyId = “Policy 3” rule-combining-algorithm=”only-one-applicable”>
        // TCSEC MAC Policy Specification //
   <Target> /* Policy applies to all subjects with clearance levels – Top-Secret, Secret, or
              Unclassified and resources with classification levels – Top-Secret, Secret, or
              Unclassified for both “read” and “write” actions */
     /* :Attribute-Category   : Attribute ID   : Attribute Value */
          :access-subject     :Clearance       :Top-Secret
          :access-subject     :Clearance       :Secret
```

```
            :access-subject     :Clearance      :Unclassified
            :resource           :Classification :Top-Secret
            :resource           :Classification :Secret
            :resource           :Classification :Unclassified
            :action             :action-id      :read
            :action             :action-id      :write
    </Target>

/* Rule 1 and Rule 2 apply to permissible and non-permissible "reads" */
<Rule RuleId = "Rule 1" Effect="Permit">
        <Target>
            /* :Attribute-Category   : Attribute ID    :Attribute Value */
                :action                :action-id        :read
         </Target>
        <Condition>
            Function: string-greater-or-equal
            /* :Attribute-Category   :Attribute ID
                :access-subject      :Clearance
                :resource            :Classification
        </Condition>
    </Rule>
    <Rule RuleId = "Rule 2" Effect="Deny">
        <Target>
            /* :Attribute-Category   :Attribute ID   : Attribute Value */
                :action                :action-id        :read
         </Target>
        <Condition>
            Function: string-less
            /* :Attribute-Category   : Attribute ID
                :access-subject      :Clearance
                :resource            :Classification
        </Condition>
    </Rule>

      /* Rule 3 & Rule 4 apply to permissible and non-permissible "writes" */
    <Rule RuleId = "Rule 3" Effect="Permit">
        <Target>
            /* :Attribute-Category   : Attribute ID   : Attribute Value */
                :action                :action-id        :write
         </Target>
        <Condition>
            Function: string-less-or-equal
            /* :Attribute-Category   : Attribute ID
                :access-subject      :Clearance
                :resource            :Classification
        </Condition>
```

```
        </Rule>
        <Rule RuleId = "Rule 4" Effect="Deny">
            <Target>
                /* :Attribute-Category   : Attribute ID   : Attribute Value */
                    :action                 :action-id        :write
            </Target>
            <Condition>
                Function: string-greater
                /* :Attribute-Category   : Attribute ID
                    :access-subject      :Clearance
                    :resource            :Classification
            </Condition>
        </Rule>
    </Policy>
```

Assuming that a user was assigned to Top Secret, Secret, or Unclassified, Policy3 would indeed enforce the TCSEC MAC policy, but would prevent a user from ever writing to a resource below the user's clearance level.

Now consider NGAC's specification of the same MAC policy, shown in Figure 9, where we assume users (not shown) are directly assigned to Top Secret or Secret (on the right side) and objects are directly assigned to Top Secret or Secret (on the left side).

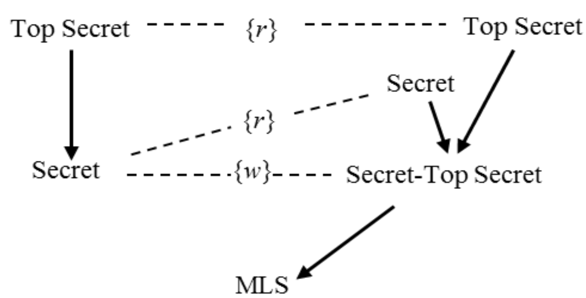

Figure 9: NGAC's Partial Expression of TCSEC MAC

The assignments and associations of the graph specify Top Secret users can read and write Secret and Top Secret objects, and Secret users can read Secret objects and write to Secret and Top Secret objects. Note that the assignments and associations alone do not prevent the leakage of data of a higher classification to a lower classification. With the following two obligations, NGAC can prevent illicit leakage of data, while allowing the user the full set of capabilities permitted by the assignments and associations. In other words, a user could read Top Secret data and write to Secret data in the same session, but through two different processes.

(1) when process p reads $o \rightarrow^{+} \mathit{TopSecret}$ **do** create p-deny(p, $\{w\}$, $\neg$*Top Secret*);

(2) when process p reads $o \rightarrow^{+} \mathit{Secret}$ **do** create p-deny(p, $\{w\}$, $\neg$*Secret-Top Secret*).

The first obligation specifies: when a process reads an object contained in Top Secret, deny the process from writing to any object outside the Top Secret (object attribute) container. Similarly, the second obligation specifies: when a process reads an object contained in the Secret-Top Secret container, deny the process from writing to any object outside the Secret-Top Secret container.

Without support for confinement, XACML is arguably incapable of enforcement of a wide variety of policies. These confinement-dependent policies include some instances of RBAC, e.g., "only doctors can read medical records", ORCON and Privacy [10], e.g., "I know who can currently read my data or personal information", or conflict of interest [4], e.g., "a user with knowledge of information within one dataset cannot read information in another dataset". Through imposing process level controls in conjunction with obligations, NGAC has shown [7] support for these and other confinement-dependent MAC controls.

Although XACML and NGAC have the ability to combine policies, their motivations are different. XACML's motivation is to resolve conflicts. That is, policies and rules may have different Effects (Permit or Deny), which must be resolved during evaluation by selectively applying one of several combining algorithms. NGAC's motivation is to ensure the adherence of combinations of multiple policies when computing a decision (e.g., DAC and RBAC).

5.3 Operational Efficiency

While XACML and NGAC are similar in that they selectively identify and evaluate policies and conditions that pertain to a request, they differ significantly in their approach. An XACML request is a collection of attribute name-value pairs for the subject (user), action, resource, and environment that must be translated to an XACML canonical form for PDP consumption. XACML identifies applicable policies and rules within policies by matching attributes to Targets. The entire process involves collecting attributes and matching Target conditions over all policies (trusted and untrusted access policies) and all rules in applicable policies, issuing administrative requests (for determining a chain of trust for applicable untrusted access policies). If the attributes are not sufficient for the evaluation of an applicable policy or rule, the PDP may search for additional attributes. The access process involves searching at least two data stores (PIP and PRP). The PDP evaluates each applicable rule in a policy and applies a combining algorithm in rendering a policy level decision. The process continues over all applicable policies and renders an ultimate decision by applying a combining algorithm over the evaluation results of the policies. The PDP response is converted from its canonical form back to the native form.

NGAC is inherently more operationally efficient. In response to an access request, a decision is computed using access control data stored in one database. NGAC identifies relevant policies and attributes directly through assignment relations. Like XACML, NGAC combines policies. However, unlike XACML, it does not compute and then combine multiple local decisions, but rather takes multiple policies into consideration when determining the existence of an appropriate privilege. If such a privilege does exist and no exceptions (prohibitions) exist, the request is granted, otherwise it is denied. Like policies and attributes, prohibitions are found through relations and not search. NGAC does not include a context handler for converting requests and decisions to and from its canonical form or for retrieving attributes. Although considered a component of its access control process, obligations do not come into play until after a decision has been rendered and data has been successfully altered or consumed.

5.4 Attribute and Policy Management

XACML and NGAC both offer a delegation mechanism in support of decentralized administration of access policies. Both allow an authority (delegator) to delegate all or parts of its own authority or someone else's authority to another user (delegate). Unlike NGAC, XACML's delegation method is a partial solution. It is dependent on trusted and untrusted policies, where trusted policies are assumed valid, and their origin is established outside the delegation model. XACML enables policy statements to be written by multiple writers. Although XACML facilitates the independent writing, collection, and combination of policy components, XACML does not describe any normative way to coordinate the creation and modification of policy components among these writers. NGAC enables a systematic approach to the creation of administrative responsibilities. The approach begins with a single administrator that can create and delegate administrative capabilities to include further delegation authority to intermediate administrators. The process ends with users with data service, policy, and attribute management capabilities.

Although one could imagine a means of administering attributes through the use of XACML policies, in practice the creation of attribute values and subject and resource assignments to those attributes is typically performed in different venues without any notion of coordination or governance.

Because XACML is implemented in XML, it inherits XML's benefits and drawbacks. The flexibility and expressiveness of XACML, while powerful, make the specification of policy complex and verbose [12]. Applying XACML in a heterogeneous environment requires fully specified data type and function definitions that produce a lengthy textual document, even if the actual policy rules are trivial. In general, platform-independent policies expressed in an abstract language are difficult to create and maintain by resource administrators [14]. Unlike XACML, NGAC is a relations-based standard, which avoids the syntactic and semantic complexity in defining an abstract language for expressing platform-independent policies [12]. NGAC policies are expressed in terms of configuration elements that are maintained at a centralized point and typically rendered and manipulated graphically. For example, to describe hierarchical relations between attributes, NGAC requires only the addition of links representing assignment relations between them; in XACML, relations need to be inserted in precise syntactic order.

NGAC's ability to express policies graphically aids in the management of policy expressions; administrators can "see" how the managed attributes are related to each other, as well as the policies under which the attributes are covered.

XACML does not allow policies to be modified by ordinary users. NGAC manages its access control data (policies and attributes) through a standard set of administrative operations, applying the same PEP interface and decision making function it uses for accessing its objects (resources). In other words, NGAC does not make a distinction between ordinary users and administrators; users possess varying flavors of capabilities to access resource objects and access control data objects. On one extreme a user may have only capabilities for administering a mandatory policy, and denied the ability to provision their access to resources governed by that policy. On the other extreme users may have total control over their own data and be responsible for setting up their own policies. Examples of the latter extreme include social networking, messaging, and calendar application capabilities.

XACML's ability to specify policies as conditions provides policy expression efficiency. Consider the NGAC expression, shown in Figure 7, of the equivalent XACML Policy1 specified in Section 3.4. NGAC expresses the policy using five association relations, while XACML uses just three rules. Note that as the number of Wards that are considered by the policy increases, so will the number of NGAC association relations, but the number of XACML rules will always remain the same. Recognize that for this policy, the number of attribute assignments is the same for XACML and NGAC. On the other hand, for some policies, the number of XACML attribute assignments can far exceed those necessary for an NGAC equivalent policy. Consider the TCSEC MAC Policy expressed using XACML rules and NGAC relations specified in Section 5.2. Note that under the NGAC configuration there is no need to directly specify policy or attributes regarding uncleared users or unclassified objects. More significantly, NGAC requires far fewer attribute assignments. For the XACML TCSEC MAC policy to work, all resources are required to be assigned to Unclassified, Secret, or Top Secret attributes. For the NGAC TCSEC MAC policy to work, only objects that are actually classified are required to be assigned to Secret or Top Secret attributes.

5.5 Administrative Review and Resource Discovery

A desired feature of access controls is review of capabilities of a user/subject and access control entries of an object/resource [15], [11]. This feature is also referred to as "before the fact audit" and resource discovery. "Before the fact audit" has been suggested by some as one of RBAC's most prominent features [18], and includes being able to review the capabilities of a user or the consequences of assigning a user to a role. It also includes the capability for a user to discover or see accessible resources. Being able to review the access control entries of an object/resource is equally important. Who are the users/subjects that can access this object/resource and what are the consequences of assigning an object/resource to an attribute or deleting an assignment?

NGAC supports efficient algorithms for both per-user and per-object review. Per-object review of access control entries (*u*, *op*), where *u* is a user and *op* is an operation, is clearly not as efficient as a pure access control list (ACL) mechanism, and per-user review of capabilities (*op*, *o*), where *op* is an operation and *o* is an object, is not as efficient as that of RBAC. However, this is due to NGAC's consideration of conducting review in a multiple policy class environment. NGAC can efficiently support both per-object and per-user reviews of combined policies, where RBAC and ACL mechanisms can do only one type of review efficiently. Rule-based mechanisms, such as XACML, although able to combine policies, cannot do either efficiently [7]. This is because determining an authorization for a subject to perform an action on a resource can only be determined by issuing a request. In other words, there exists no method of determining the authorization state without testing all possible decision outcomes.

Appendix A—Acronyms

Selected acronyms and abbreviations used in this document are defined below.

ABAC	Attribute Based Access Control
ACL	Access Control List
ANSI/INCITS	American National Standards Institute/International Committee for Information Technology Standards
API	Application Programming Interface
DAC	Discretionary Access Control
EPP	Event Processing Point
FISMA	Federal Information Security Modernization Act
IR	Interagency Report
IT	Information Technology
ITL	Information Technology Laboratory
MAC	Mandatory Access Control
NGAC	Next Generation Access Control
NGAC-FA	Next Generation Access Control Functional Architecture
NGAC-GOADS	Next Generation Access Control Generic Operations and Abstract Data Structures
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OMB	Office of Management and Budget
ORCON	Originator Controlled
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PM	Policy Machine
PRP	Policy Retrieval Point
RAP	Resource Access Point
RBAC	Role-Based Access Control
RS	Resource Server
SAML	Security Assertion Markup Language
SOA	Service Oriented Architecture
SP	Special Publication
TCSEC	Trusted Computer Security Evaluation Criteria
XACML	Extensible Access Control Markup Language
XML	Extensible Markup Language

Appendix B—References

[1] Information technology – Role-Based Access Control (RBAC), INCITS 359-2004, American National Standard for Information Technology, American National Standards Institute, 2004.

[2] Information technology - Next Generation Access Control - Functional Architecture (NGAC-FA), INCITS 499-2013, American National Standard for Information Technology, American National Standards Institute, March 2013.

[3] D. Bell and L. La Padula. Secure computer systems: unified exposition and MULTICS. Report ESD-TR-75-306, The MITRE Corporation, Bedford, Massachusetts, March 1976.

[4] D.F.C. Brewer and M.J. Nash, "The Chinese Wall Security Policy," *1989 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1-3, 1989, pp. 206-214. http://dx.doi.org/10.1109/SECPRI.1989.36295 [accessed 11/15/15]

[5] DoD Computer Security Center, Trusted Computer System Evaluation Criteria (December 1985).

[6] D.F. Ferraiolo, S.I. Gavrila, V.C. Hu, and D.R. Kuhn, "Composing and Combining Policies Under the Policy Machine," *Tenth ACM Symposium on Access Control Models and Technologies (SACMAT '05)*, Stockholm, Sweden, 2005, pp. 11-20. http://dx.doi.org/10.1145/1063979.1063982 [accessed 11/15/15] or https://csrc.nist.gov/staff/Kuhn/sacmat05.pdf [accessed 11/15/15]

[7] D.F. Ferraiolo, V. Atluria, and S.I. Gavrila, "The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 412-424, April 2011. http://dx.doi.org/10.1016/j.sysarc.2010.04.005 [accessed 11/15/15]

[8] D. Ferraiolo, S. Gavrila, and W. Jansen, National Institute of Standards and Technology (NIST) Internal Report (IR) 7987 Revision 1, "Policy Machine: Features, Architecture, and Specification," October 2015. http://nvlpubs.nist.gov/nistpubs/ir/2015/NIST.IR.7987r1.pdf [accessed 11/15/15]

[9] D. Ferraiolo, S. Gavrila, and W. Jansen, "On the Unification of Access Control and Data Services," in Proceedings of the IEEE 15th International Conference of Information Reuse and Integration, 2014, pp. 450 – 457. http://csrc.nist.gov/pm/documents/ir2014_ferraiolo_final.pdf [accessed 11/15/15]

[10] R. Graubart, On the need for a third form of access control, in: Proceedings of the National Computer Security Conference, 1989, pp. 296 –304.

[11] V.C. Hu, D.F. Ferraiolo, and D.R. Kuhn, National Institute of Standards and Technology (NIST) Interagency Report (IR) 7316, “Assessment of Access Control Systems,” September 2006. http://csrc.nist.gov/publications/nistir/7316/NISTIR-7316.pdf [accessed 11/15/15]

[12] V. C. Hu, D.F. Ferraiolo, and K. Scarfone, Access Control Policy Combinations for the Grid Using the Policy Machine, Cluster Computing and the Grid, 2007, pp. 225-232.

[13] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, National Institute of Standards and Technology (NIST) Special Publication (SP) 800-162, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, January 2014. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf [accessed 11/15/15]

[14] M. Lorch et al, “First Experience Using XACML for Access Control in Distributed Systems, ACM Workshop on XML Security, Fairfax, Virginia, 2003.

[15] *A Guide to Understanding Discretionary Access Control in Trusted Systems*, NCSC-TG-003, Version-1, National Computer Security Center, Fort George G. Meade, Maryland, USA, September 30, 1987, 29 pp. http://csrc.nist.gov/publications/secpubs/rainbow/tg003.txt [accessed 11/15/15]

[16] XACML Profile for Role Based Access Control (RBAC), Committee Draft 01, February 2004.

[17] The eXtensible Access Control Markup Language (XACML), Version 3.0, OASIS Standard, January 22, 2013. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf [accessed 11/15/15]

[18] 2010 Economic Analysis of Role-Based Access Control, RTI Number 0211876, Research Triangle Institute, December 2010.

[19] R. Simon, M. Zurko, Separation of duty in role based access control environments, in: Proc. of the New Security Paradigms Workshop, 1997.

[20] Information technology – Next Generation Access Control – Generic Operations and Data Structures, INCITS 526, American National Standard for Information Technology, American National Standards Institute, to be published.

Appendix C—XACML 3.0 Encoding of Medical Records Access Policy

/* This policy pertains to Medical Record (Read or Write) Access by users with role “Doctor” or “Intern”. Rule 1 denies access if the WardAssignment of the doctor or intern does not match the WardLocation of the patient. Rule 2 denies write access to intern unconditionally. Rule 3 permits access if the subject is a doctor and the PatientStatus is Critical without any other conditions. */

```
<Policy PolicyId=”Medical-Record-Access-by-Doctors-and-Interns”
            RuleCombiningAlgId = “permit-overrides”>

<Target> /* Policy Target covers all subjects with Doctor or Intern role, resources with medical-
records as Resource-id, and actions either read or write */

  <AnyOf>
    <AllOf> /* Specifying the subject match – subjects with role-id equal to Doctor or Intern */
      <Match MatchId="string-equal"> /* Subject role = Doctor */
              <AttributeValue> Doctor </AttributeValue>
              <AttributeDesignator Category=”access-subject” AttributeId=”role-id”/>
      </Match>
   <AllOf>
   <AllOf> /* Specifying the subject match – subjects with role-id equal to Doctor */
      <Match MatchId="string-equal"> /* Subject role = Intern */
              <AttributeValue> Intern </AttributeValue>
              <AttributeDesignator Category=”access-subject” AttributeId=”role-id”/>
      </Match>
   <AllOf>
</AnyOf>

  <AnyOf>
    <AllOf> /* Specifying the resource match – resource with resource-id equal to medical-
                    records */
      <Match MatchId="string-equal">
              <AttributeValue> medical-records</AttributeValue>
              <AttributeDesignator Category=”resource” AttributeId=”resource-id”/>
      </Match>
   </AllOf>
</AnyOf>

 <AnyOf> /* Specifying action match – action with either read or write value */
    <AllOf>  /* read action */
      <Match MatchId="string-equal">
              <AttributeValue> read</AttributeValue>
              <AttributeDesignator Category=”action” AttributeId=”action-id”/>
      </Match>
   </AllOf>
   <AllOf>  /* write action */
      <Match MatchId="string-equal">
```

```
            <AttributeValue> write</AttributeValue>
            <AttributeDesignator Category=”action” AttributeId=”action-id”/>
      </Match>
    </AllOf>
   </AnyOf>
</Target>

<Rule RuleId=”Rule 1”
        Effect=”Deny”> /* denial of access to medical record for all subjects if the patient is not
                                in the same ward to which the doctor or intern is assigned */
    <Condition>
        <Apply FunctionId=”string-not-equal”>
          <Apply FunctionId=”string-one-and-only”>
              <AtributeDesignator Category=”access-subject” AttributeId=”WardAssignment”>
          </Apply>
          <Apply FunctionId=”string-one-and-only”>
              <AtributeSelector Category=”resource”
                Path=”medical-records/patient/WardLocation/text( )”/>
          </Apply>
      </Condition>
   </Rule>

  <Rule RuleId=”Rule 2”
        Effect=”Deny”> /* unconditional denial of write access to Interns */
    <Condition>
        <Apply FunctionId=”string-equal”>
          <Apply FunctionId=”string-one-and-only”>
            <AttributeValue> Intern</AttributeValue>
            <AttributeDesignator Category=”access-subject” AttributeId=”role-id”/>
          </Apply>
          <Apply FunctionId=”string-one-and-only”>
              <AttributeValue> write</AttributeValue>
              <AtributeDesignator Category=”action” AttributeId=”action-id”>
          </Apply>
      </Condition>
   </Rule>

  <Rule RuleId=”Rule 3”
        Effect=”Permit”> /* unconditional access to medical records for doctor if the patient status
                                is critical irrespective of the location of the patient */
    <Condition>
        <Apply FunctionId=”and”> /* combines subject role value and patient status value */

            <Apply FunctionId=”string-one-and-only”> /* retrieves the subject role */
              <AttributeValue> doctor</AttributeValue>
              <AttributeDesignator Category=”access-subject” AttributeId=”role-id”/>
            </Apply>
```

```
            <Apply FunctionId="string-equal"> /* looks for medical records where patient
                                                 status is critical */
             <Apply FunctionId="string-one-and-only">
               <AttributeSelector Category="resource"
                 Path="medical-records/patient/PatientStatus/text( )"/>
             </Apply>
            <AttributeValue>Critical</AttributeValue>
           </Apply>
       </Apply>
     </Condition>
   </Rule>
</Policy>
```